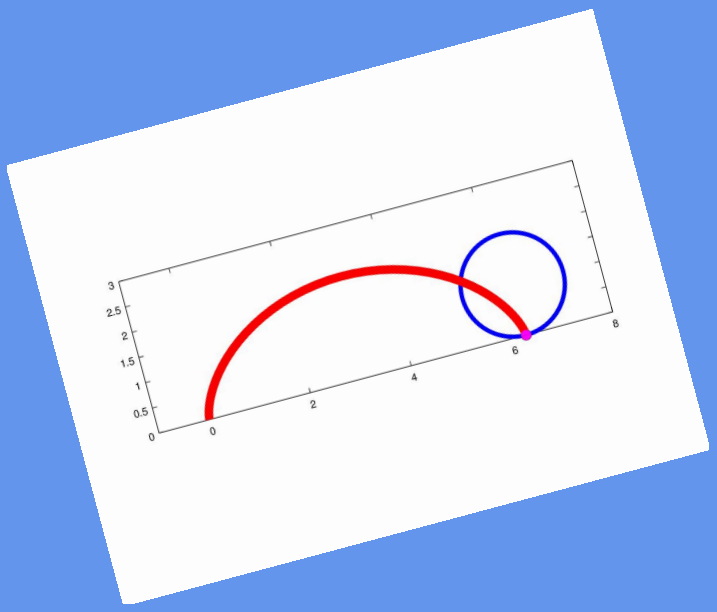
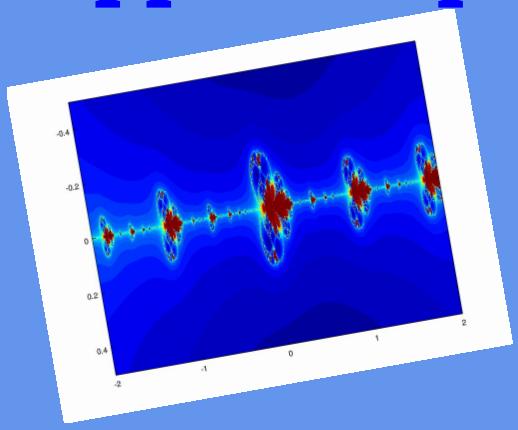
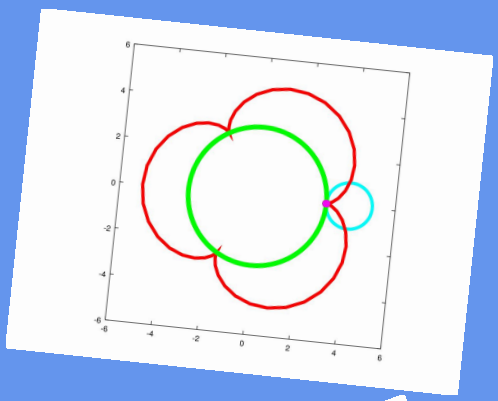
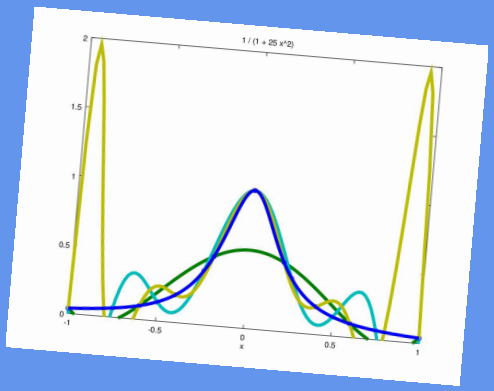


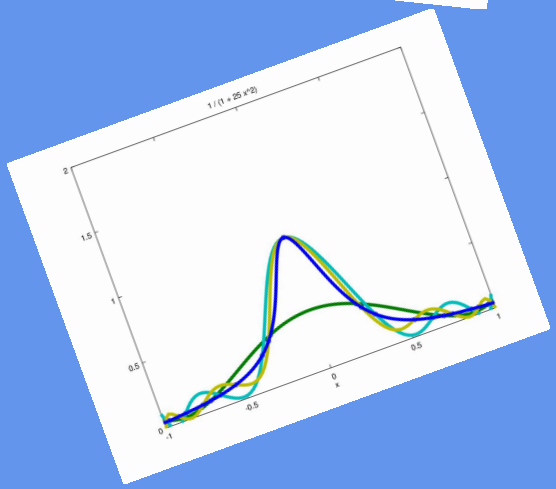
Annamaria Mazzia

Appunti sparsi su Octave









Dispensa aggiornata in Febbraio 2017

OGNI UOMO DOVREBBE GUARDARE DENTRO DI SÈ PER IMPARARE IL SIGNIFICATO DELLA VITA. NON È QUALCOSA CHE SI SCOPRE: È QUALCOSA CHE SI DEVE MODELLARE.

ANTOINE DE SAINT-EXUPÉRY

Indice

1 *Iniziamo* 3

2 *Entriamo in ambiente Octave* 11

3 *Tipi di dati* 21

4 *Proposizioni e predicati logici* 31

5 *Programmi, codici e pseudocodici* 37

6 *Matrici e vettori (parte prima)* 63

7 *Funzioni matematiche e grafici* 69

8 *Metodi iterativi per zeri di funzione* 85

9 *Interpolazione e approssimazione di dati* 97

10 *Matrici e vettori (parte 2)* 107

11 *Integrazione numerica* 125

12 *Informazioni utili* 131

Bibliografia 135

Perchè queste pagine

Colui che vuole viaggiare felice deve viaggiare leggero.
Antoine de Saint-Exupéry

DA QUALCHE ANNO al corso di Calcolo Numerico per gli studenti di Ingegneria dell'Energia propongo come linguaggio di programmazione l'ambiente MATLAB®.

Le lezioni di laboratorio in Aula Taliercio sono sempre troppo poche rispetto a quello che si vorrebbe fare e il libro che ho già scritto (Laboratorio di Calcolo Numerico con MATLAB® Octave) è poco letto e poco usato 🙄.

Gli errori che vedo fare nei compiti di programmazione 😱mi hanno indotto ad iniziare a scrivere questo tutorial, sperando che gli studenti potranno usare queste pagine come aiuto per imparare a lavorare meglio in MATLAB®.

Parlerò di Octave, perchè è un ambiente free rispetto a MATLAB® ma parlerò solo degli aspetti comuni (che saranno quelli che ci interessano). Le differenze, dove occorre, saranno ben evidenziate. Vorrei sfatare in questo modo qualche commento di qualche studente che dice che non riesce a lavorare in Octave perchè è molto diverso da MATLAB® ‼️

Buon lavoro 😀

Padova, Febbraio 2017

Annamaria Mazzia

P.S. Per migliorare sempre più questo tutorial, se avete dubbi su alcune parti o trovate errori, di stampa e non, potete gentilmente segnalarmi le vostre perplessità all'indirizzo email:
`annamaria.mazzia CHIOCCIOLA unipd.it`

P.P.S. La prima bozza di questo tutorial è terminata a pochi giorni dall'inizio delle lezioni, quindi può essere che ci sia qualche errore di stampa e qualche svista!

1

Iniziamo

Qualche cosa si era rotta nel motore, e siccome non avevo con me nè un meccanico, nè dei passeggeri, mi accinsi da solo a cercare di riparare il guasto. Era una questione di vita o di morte, perchè avevo acqua da bere soltanto per una settimana.
Antoine de Saint-Exupéry

Prima di entrare nel vivo dell'ambiente di programmazione, spendiamo qualche pagina per parlare di algoritmi, programmi, sottoprogrammi e diagrammi di flusso.

- Un'algoritmo rappresenta la strategia di soluzione di un problema, un procedimento di calcolo che ci permette di risolvere il problema stesso.
- Un programma rappresenta la *traduzione* di un algoritmo in un determinato linguaggio di programmazione. Se all'interno del programma alcune parti sono ripetitive, queste sono *affidate* a dei sottoprogrammi.
- Il diagramma di flusso, invece, rappresenta l'approccio visivo che traduce l'algoritmo prima ancora di passare ad un linguaggio di programmazione.

Facciamo degli esempi su algoritmi e diagrammi di flusso, in modo da poter poi entrare nel vivo della programmazione.

1.1 Un algoritmo

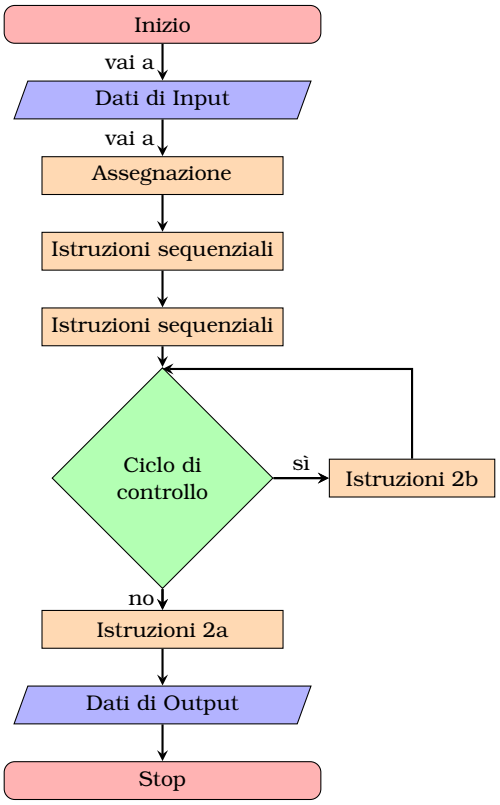
Avete presente il gioco del lancio della monetina per vedere se esce *testa o croce* e prendere una decisione? Bene. Possiamo vedere questo *gioco* come un algoritmo per effettuare una particolare scelta (ad esempio: se esce testa andiamo al cinema, se esce croce si va in pizzeria). L'algoritmo si può vedere in questo modo.

- Dato di ingresso: la monetina
- Istruzioni da eseguire: lanciare in aria la monetina
- Controllare se la faccia della monetina che vediamo sia quella di *testa* o quella di *croce*
- Il dato di uscita è la faccia della monetina lanciata.

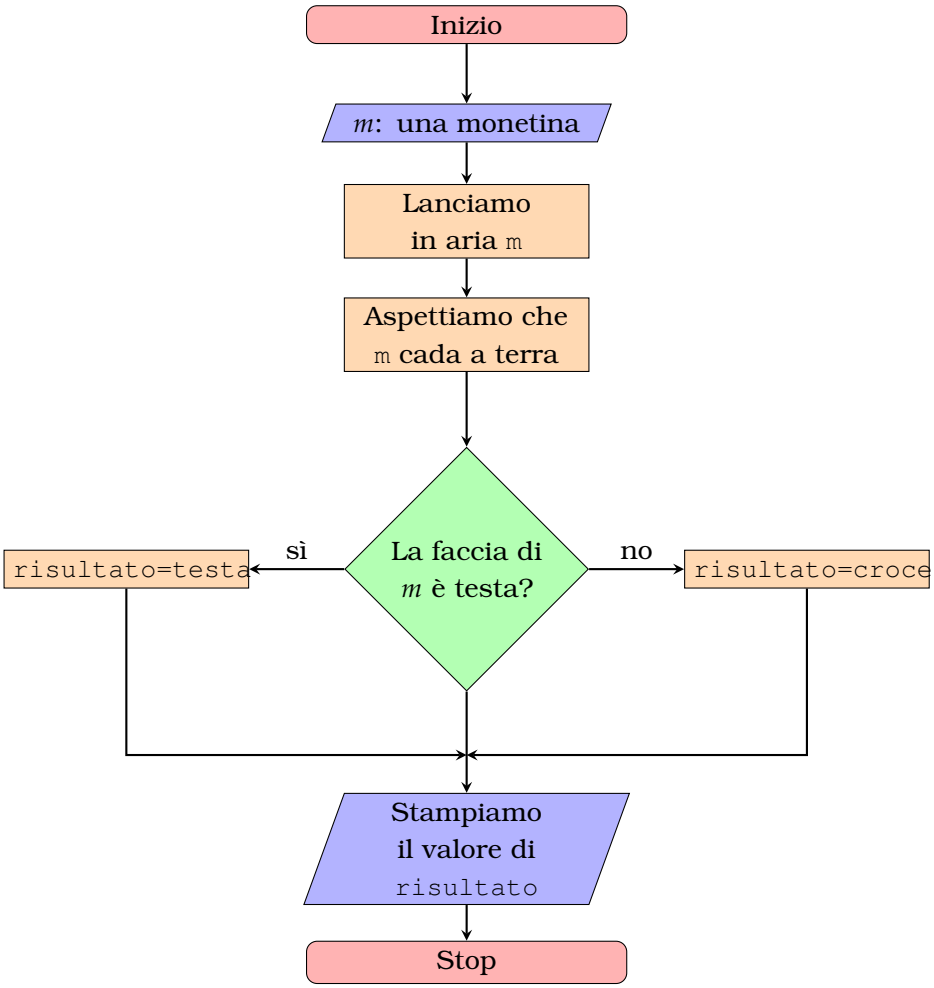
Queste operazioni che abbiamo detto si traducono in alcune istruzioni fondamentali (e che useremo sempre nei nostri programmi):

- istruzioni di assegnazione: assegneremo ad una variabile, occupando un certo spazio nella memoria del calcolatore, il valore di una certa quantità;
- istruzioni di lettura dei dati: avremo da leggere dei dati di ingresso che dovremo assegnare a determinate variabili;
- istruzioni di scrittura: faremo scrivere dei dati che vogliamo conoscere, come, ad esempio, i risultati del nostro programma;
- istruzioni sequenziali, da eseguire una dopo l'altra;
- istruzioni mediante cicli di controllo (del tipo *if*, *for*, *while*), da eseguire in base al controllo di una proposizione logica: possiamo avere istruzioni di tipo condizionale oppure istruzioni di tipo iterativo;
- istruzioni per terminare l'esecuzione del programma.

Vediamo un esempio di diagramma di flusso, con generiche istruzioni.



Proviamo a tradurre in diagramma di flusso il problema del *testa o croce* (tranquilli: arriveremo a fare diagrammi e algoritmi più seri! 😄‼).

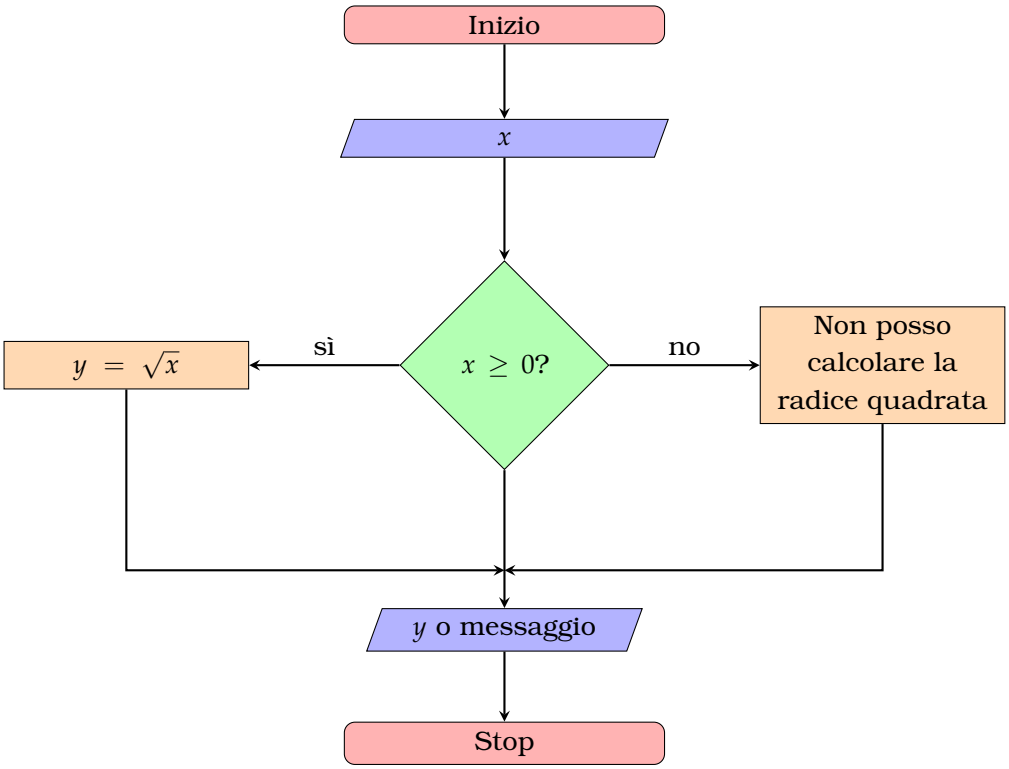


Nel diagramma di flusso che abbiamo scritto appare un'istruzione posta sotto forma di domanda: in base alla risposta eseguiremo un'istruzione piuttosto che un'altra. Rispetto alle istruzioni sequenziali, da eseguire una dopo l'altra, quelle che vanno eseguite in base ad una condizione sono le più difficili da gestire. Perciò ora perderemo un po' di tempo per capire le istruziono non sequenziali.

1.2 La struttura condizionale

Partiamo da alcuni esempi (però questa volta con dei numeri 😀).

Se vogliamo calcolare la radice quadrata positiva di un numero (e vogliamo lavorare con la radice quadrata vista come una funzione reale), dobbiamo controllare il segno del numero. Sappiamo infatti che la funzione radice quadrata, come funzione reale, è definita per $x \geq 0$ (altrimenti si entra nel campo dei numeri complessi). Quindi se $x \geq 0$ allora possiamo calcolare $y = \sqrt{x}$. Avete notato il `se`? Abbiamo una proposizione condizionale! In base ad una condizione noi possiamo fare qualcosa oppure no.



Se $x \geq 0$ calcolo $y = \sqrt{x}$, altrimenti stampo un messaggio di errore.

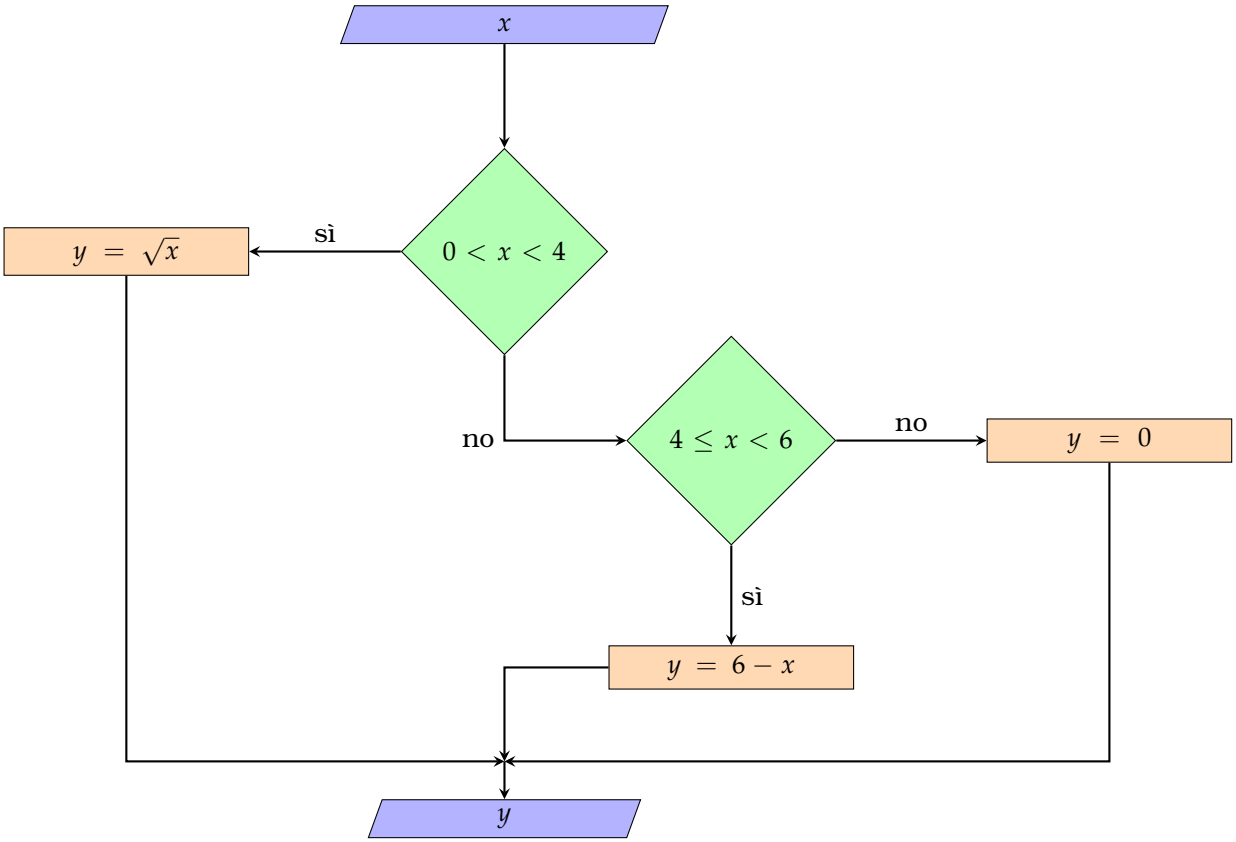
Ricordiamo bene: *Se vale una certa condizione, facciamo qualcosa, altrimenti facciamo qualche altra cosa.*

Se è vera una proposizione logica, eseguiamo determinate istruzioni, se non è vera eseguiamo altre istruzioni.

Questo che abbiamo ora descritto è il ciclo `if`.

Possiamo arricchire il ciclo aumentando i `se`:

Se $0 < x < 4$ considero $y = \sqrt{x}$, se $4 \leq x < 6$ prendo $y = 6 - x$, altrimenti (cioè se non sono vere le condizioni precedenti) allora $y = 0$.



Osserviamo come da ogni blocco dove è posta una condizione noi possiamo scegliere una sola strada, in base alla risposta che viene data. Possiamo arricchire quanto vogliamo la struttura del ciclo `if` aggiungendovi altre condizioni. E possiamo anche ridurre la struttura deil ciclo `if` dicendo soltanto che se è vera una certa condizione facciamo qualcosa. E se non è vera? Non facciamo niente! Notate che non c'è nessun *altrimenti* nella frase scritta prima?

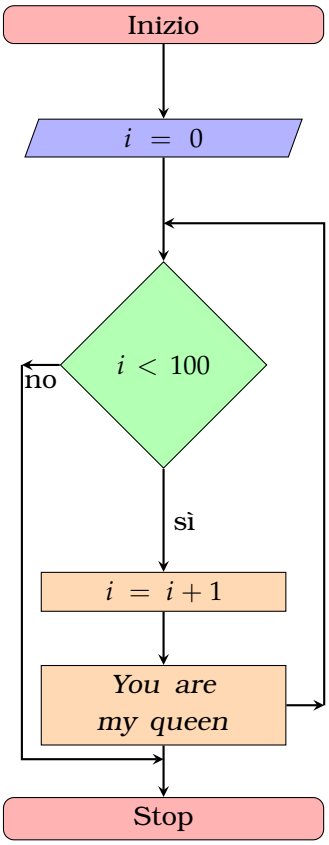
1.3 Ciclo for

Un'altra struttura molto importante è il cosiddetto ciclo `for` che serve per ripetere determinate istruzioni un certo numero di volte o per determinati valori di una particolare variabile.

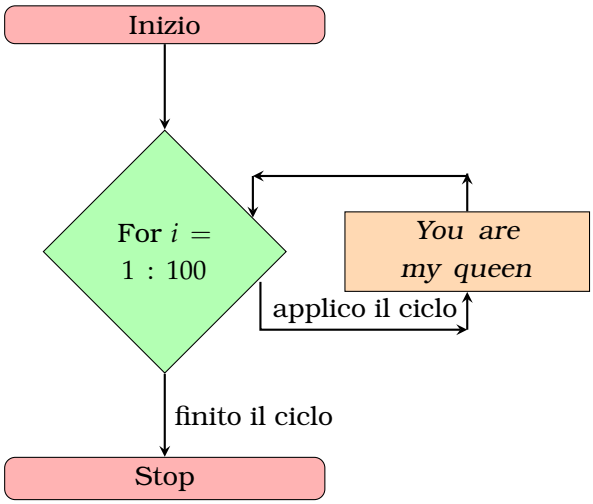
Un esempio (non matematico) 😄: Un poeta piuttosto eccentrico vorrebbe stupire la sua amata 💔scrivendole (al computer) una lettera d'amore e vorrebbe scrivere 100 volte *You are my queen!*.

Fare copia e incolla della stessa frase 100 volte può essere una soluzione, ma è molto più semplice eseguire poche righe di un programma che ci permettano di scrivere 100 volte la stessa frase facendo uso di un ciclo `for`.

L'algoritmo da tradurre è: per 100 volte scrivi questa frase. Come diagramma di flusso abbiamo



Rispetto al ciclo `if`, in questo diagramma di flusso troviamo una freccia che esce dalla condizione $i < 100$ e una freccia che rientra nella condizione stessa, proprio perchè dobbiamo eseguire alcune istruzioni più e piú volte, in questo caso fino a quando la variabile i non diventa uguale a 100. La variabile i ci permette, quindi, di scrivere 100 volte *You are my queen*. Quando impareremo ad usare il ciclo `for`, vedremo che molti di questi passaggi saranno fatti in modo nascosto, e il diagramma di flusso si potrà riscrivere nel modo seguente



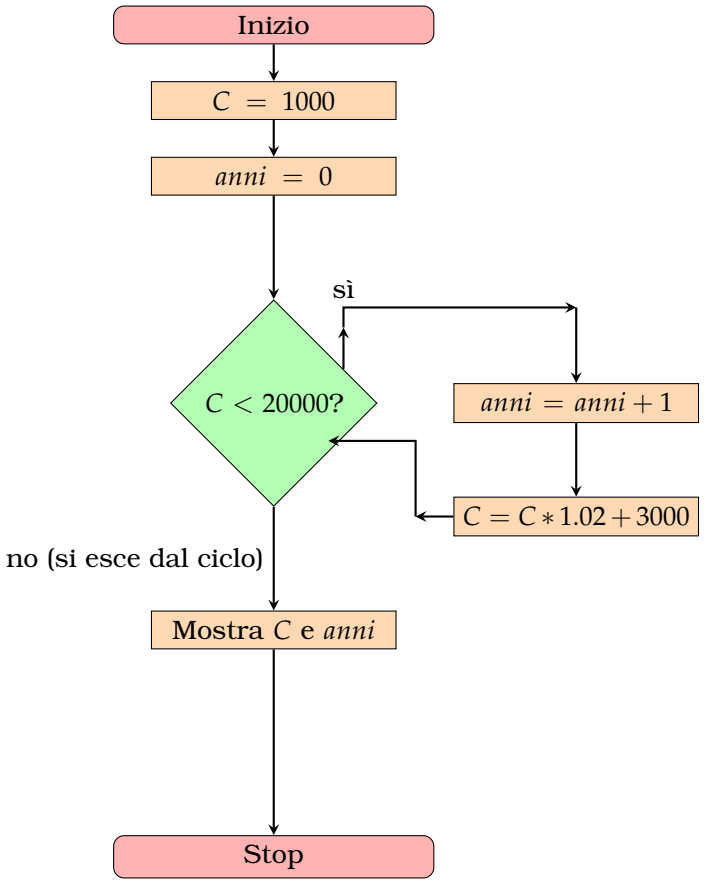
1.4 CICLO WHILE

Tra le strutture più importanti con cui prendere familiarità si trova il ciclo `while`, che ci permette di eseguire determinate istruzioni fino a quando risulta verificata una determinata condizione.

Un esempio: lo studente Pertinace, oltre a frequentare l'Università, fa dei lavoretti saltuari avendo come obiettivo l'acquisto di una moto da cross 🚵. Dal momento che non sa quando riuscirà a comprarsela, vuole capire in quanto tempo potrà mettere da parte 20.000 euro sul suo conto corrente in banca 💸. Ha già un deposito di 1000 euro e ha notato che alla fine di ogni anno può mettere da parte 3000 euro. Il tasso di interesse sul suo conto di deposito è del 2%. La formula per calcolare gli interessi sul suo conto alla fine di ogni anno è data da: $Capitale * Tassodiinteresse$. Come fare a capire in quanti anni arriverà a 20.000 euro?

Se partiamo da un capitale iniziale di 1000 euro, alla fine di ogni anno dobbiamo considerare che a quel capitale si deve aggiungere la somma di 3000 euro e gli interessi accumulati durante l'anno. Chiamando C il capitale, ogni anno il capitale accumulato sarà dato dalla formula $C = C + C * 2/100 + 3000 = C * 1.02 + 3000$.

Tracciamo un diagramma di flusso per capire in quanti anni Pertinace arriverà a 20.000 euro.



Partendo da $C = 1000$ al nostro anno iniziale ($anni = 0$), andiamo a controllare se $C < 20000$: se la condizione è soddisfatta vuol dire che dobbiamo ancora arrivare al nostro obiettivo, quindi deve passare un anno (incrementiamo di una unità la variabile che conta gli anni) e applichiamo la formula per vedere a quanto ammonta il capitale alla fine dell'anno. A questo punto torniamo alla condizione: ora C è un valore aggiornato, non è più quello di prima. Se vale di nuovo $C < 20000$, dobbiamo ancora raggiungere l'obiettivo e ripetiamo le operazioni precedenti. Si va avanti in questo modo fino a quando la condizione $C < 20000$ diventa falsa. In questo caso usciamo da questo ciclo e mostriamo quanti anni occorrono e quanto vale il capitale C. Questo ciclo si chiama ciclo `while` perchè eseguiamo delle istruzioni fintantochè, fino a quando, `while`... sono vere determinate condizioni (in questo caso, fino a quando $C < 20000$). Non eseguiremo più quelle istruzioni se quelle condizioni non saranno più verificate.

2

Entriamo in ambiente Octave

Se qualcuno ama un fiore, di cui esiste un solo esemplare in milioni e milioni di stelle, questo basta a farlo felice quando lo guarda. Antoine de Saint-Exupèry

Per imparare a programmare, iniziamo subito a vedere l'ambiente in cui lavoreremo, Octave. La prima cosa da fare è installare il software sul proprio computer. Le ultime versioni sono molto semplici da installare: `https://www.gnu.org/software/octave/` è il sito cui collegarsi per scaricare l'ultima versione di Octave.

2.1 Avviare Octave

Una volta installato Octave sul proprio computer, per avviarlo si deve

- (per chi ha Windows) cliccare sull'icona che compare sul Desktop
- (per chi ha Linux) aprire una finestra di terminale, scrivere `octave` e schiacciare il tasto di invio.
- (per chi ha Mac) probabilmente seguire la procedura come per Linux (dico probabilmente perchè non ho esperienza nel settore).

Nel momento in cui sto scrivendo queste pagine, sto usando la versione 4.0.3 di Octave.

Ecco cosa si presenta ai nostri occhi (si veda Figura 2.1)
La finestra che si apre è composta da più finestre:

- La `Command Window` o finestra dei comandi. Sarà la finestra con cui prenderemo maggiore familiarità perchè ci servirà sempre. Si osservi il simbolo del prompt dato da >>.
- La piccola finestra della `Current Directory` che ci permette di capire in quale directory ci troviamo. Se vogliamo

Non ho esperienza con il sistema operativo Mac e quindi non posso che rimandare alla pagina web o ai consigli che si trovano in rete per installare Octave su una macchina con Mac, ma per Windows e per sistemi GNU/Linux il procedimento per installare Octave non è affatto complicato.

Per chi ha Windows, è sufficiente scaricare il file eseguibile `octave-4.0.3-installer.exe` (o simile a questo ma con versioni più aggiornate di Octave) ed eseguirlo. Occorre avere anche java sul proprio computer (e se manca qualche altra cosa comparirà un messaggio di avvertimento sul proprio computer mentre si sta procedendo ad installare Octave). Per chi ha Linux, conviene prima vedere se si può installare Octave utilizzando pacchetti del proprio sistema operativo (Debian, Fedora,....). Altrimenti si possono scaricare i file sorgenti che si trovano nella sezione Download della pagine web di Octave.

Sulla pagina web di Octave è possibile trovare notizie storiche e manuali su Octave.

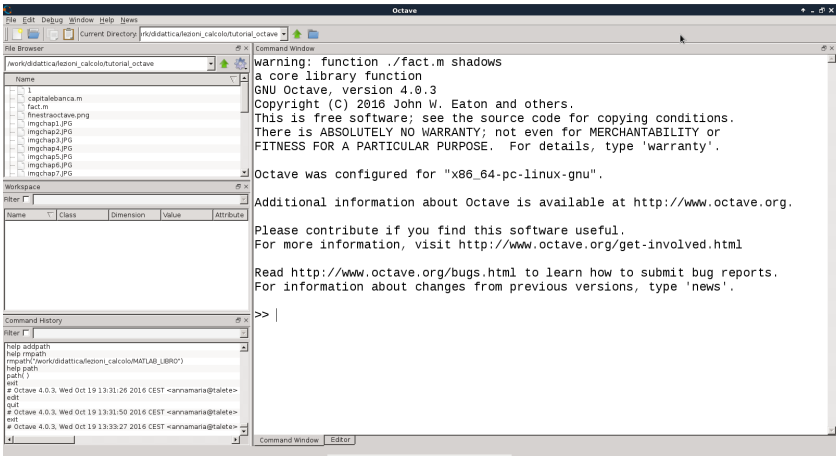


Figura 2.1: Ambiente di programmazione Octave.

cambiare directory potremo farlo facendo uso della freccia verde che vediamo alla sua destra.

- La finestra `File Browser` che ci elenca tutti i file presenti nella directory in cui stiamo lavorando.
- La finestra di `Workspace` che ci mostrerà le variabili che useremo, il loro tipo, dimensione, etc...
- La finestra della `Command History` che ci mostra tutti i comandi eseguiti sulla Command Window.

Volendo possiamo semplificare l'ambiente di lavoro modificando la scelta di ciò che compare in automatico quando si apre Octave. Possiamo fare ciò cliccando su `Window` (in alto a sinistra).

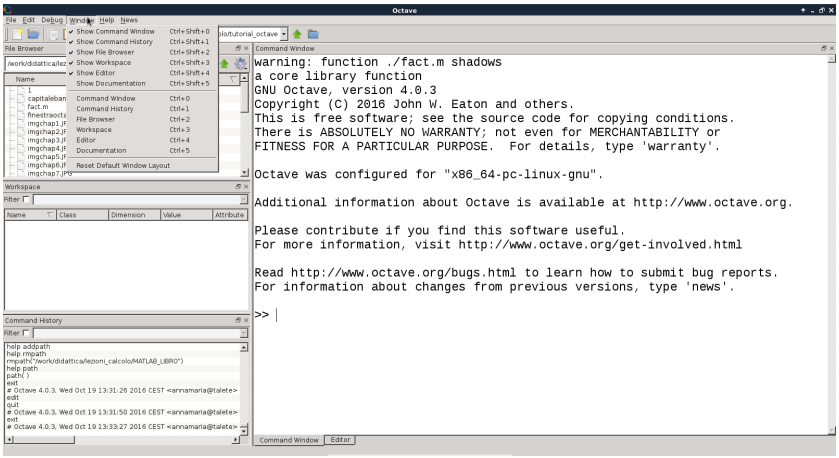


Figura 2.2: Cliccando su Window possiamo disattivare la visualizzazione di alcune finestre.

Nell'esempio di figura 2.3, sono attive la Command Window e il Workspace.

Per uscire da Octave, si può scrivere `exit` sulla Command Window oppure andando su `File` in alto a sinistra si può

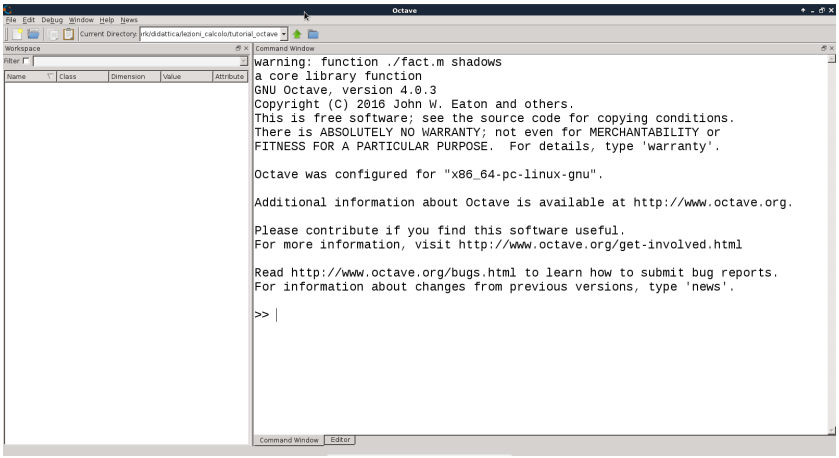


Figura 2.3: Sono attive la Command Window e il Workspace.

digitare l'Exit che si trova sul menu che si apre là, o ancora si può cliccare il simbolo a croce che si trova in alto a destra.

Per uscire è intuitivo e non stiamo a perdere altro tempo.

2.2 PRIMI PASSI

La prima cosa da fare è usare Octave come una calcolatrice. Proviamo a vedere cosa succede se, sulla Command Window, digitiamo questi comandi (schiacciando il tasto di INVIO ogni volta che passiamo all'operazione successiva):

```
3+4

512*340

3^10

10/5
```

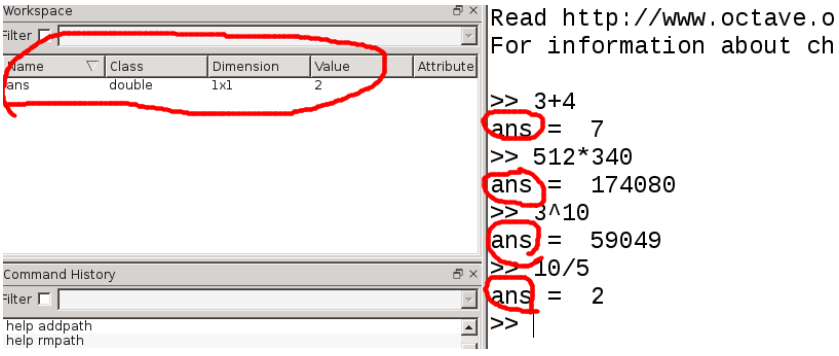


Figura 2.4: : Notare che ogni risultato viene assegnato ad una variabile chiamata `ans`. Nel Workspace troviamo informazioni sulla variabile `ans`. Notiamo anche che questa variabile viene ogni volta sovrascritta: prima vale 7, poi 174080, fino ad arrivare al valore 2 dato dall'ultima operazione che abbiamo eseguito.Vedremo tra poco in maniera più dettagliata il concetto di variabile

Naturalmente possiamo eseguire calcoli più complicati di questi e in modo molto più semplice rispetto ad una calcolatrice in quanto possiamo vedere bene ciò che stiamo scrivendo.

Per il momento stiamo lavorando con numeri. Un numero, preso singolarmente, rappresenta uno `scalare` e viene memorizzato come una variabile scalare..

Per le operazioni aritmetiche di base, abbiamo a disposizione gli operatori mostrati in Tabella 2.1.

Simbolo	Operazione	Uso
∧	elevamento a potenza: a^b	`a^b`
∗	moltiplicazione: ab	`a*b`
/	divisione a destra: $a/b = \frac{a}{b}$	`a/b`
\	divisione a sinistra: $a\backslash b = \frac{b}{a}$	`a\b`
+	addizione: $a+b$	`a+b`
−	sottrazione: $a-b$	`a-b`

Tabella 2.1: Operazioni aritmetiche

Osserviamo che esiste l'operatore di divisione che usiamo comunemente è dato dal simbolo dello `slash` /. Invece il simbolo di `backslash` \ ci dà la cosiddetta *divisione a sinistra* (e verrà utilizzato per risolvere sistemi lineari).

2.3 LE VARIABILI

Per eseguire le nostre operazioni, possiamo evitare di vedere la risposta in `ans`: `ans` sta per *answer* e contiene il risultato della nostra operazione. Abbiamo detto che si chiama *variabile*: una variabile è un simbolo utilizzato per contenere una certa quantità numerica. Dal momento che stiamo lavorando con scalari, la variabile `ans` è una variabile scalare. Ed è una variabile particolare perchè contiene il risultato più recente, dell'ultima operazione eseguita. Possiamo usare variabili definite da noi per scrivere il risultato di espressioni matematiche. Ad esempio, al posto di avere `ans=2` nell'operazione che abbiamo visto prima, potremmo assegnare il valore dell'operazione 10/5 ad una variabile che scegliamo noi, ad esempio alla variabile `a`.

```
» a=10/5
a =  2
»
```

Possiamo scegliere, come nomi da dare alle nostre variabili, tutte le lettere dell'alfabeto, sia maiuscole che minuscole, i numeri e il simbolo di underscore _ fino ad un massimo di 63 lettere! Vediamo un altro esempio

```
» a=5
a =  5
» b=12
b =  12
» c=a*b
c =  60
```

In questo caso abbiamo assegnato alle variabili `a` e `b` il valore di due scalari e poi abbiamo eseguito l'operazione di moltiplicazione tra le due variabili, traducendo quindi la formula $c = ab$.

! Ovviamente conviene chiamare le variabili con nomi che abbiano un significato rispetto a quello che stiamo facendo. `area`, `a`, `b`, `c`, `volume`, `h`. Certamente possiamo chiamare una variabile anche `AaBb_123` ma quando la rivedremo probabilmente non ci ricorderemo più a cosa serve!!! Importante è invece sottolineare che la variabile `A` non è uguale alla variabile `a`: quindi se incominciamo a lavorare con una variabile scritta in lettere minuscole, ad un certo punto non possiamo scrivere la STESSA variabile in lettere maiuscole perchè è come se introducessi una seconda variabile!!!!!

! Un errore comune è dimenticarsi di usare il simbolo `*` per fare le moltiplicazioni. Se andassimo a scrivere `c=ab` avremmo un errore!

```
» c=ab
error:  'ab' undefined near
line 1 column 3
```

Sulla scelta del nome da dare ad una variabile, bisogna prestare molta attenzione a non considerare nomi che hanno già una valenza importante in Octave perchè sono dei termini riservati per la programmazione. Si tratta, in pratica, di parole chiave che hanno il loro compito ben preciso per la programmazione! Un elenco di questi nomi è il seguente (consideriamo i più importanti): `break`, `case`, `else`, `elseif`, `end`, `for`, `function`, `if`, `otherwise`, `return`, `switch`, `while`. Se ci venisse in mente di chiamare una variabile `for`, avremmo il seguente messaggio:

```
» for=10
parse error:

  syntax error

>> for=10
       ^

»
```

Per capire se il nome che vogliamo dare ad una variabile è una parola chiave oppure no, possiamo applicare la funzione **iskeyword** al nome: se abbiamo come risposta 1, vuole dire che quella è una parola chiave e non possiamo usarla; se invece la risposta è 0, allora non abbiamo una parola chiave.

Se scriviamo solo **iskeyword** e schiacciamo il tasto di invio, abbiamo tutto l'elenco delle parole chiavi!

```
» iskeyword('for')
ans =  1
» iskeyword('a')
ans = 0
```

2.4 Precedenze sulle operazioni aritmetiche

Spendiamo ora qualche parola sull'ordine in cui vengono effettuate le operazioni, cioè sulle precedenze! Intanto le espressioni matematiche vengono valutate partendo da sinistra e andando verso destra. L'esponente ha l'ordine di precedenza più alto, poi hanno uguale grado di precedenza sia la moltiplicazione sia la divisione. In ultimo, con la stessa precedenza, troviamo l'addizione e la sottrazione. Per modificare le precedenze, occorre usare le parentesi tonde. Facciamo un esempio. Se vogliamo calcolare $10+3^25$ andremo a scrivere

```
» 10+3^2*5
ans =  55
```

Ma se vogliamo $\frac{10+3^25}{4}$ andremo a mettere le parentesi tonde:

```
» (10+3^2*5)/4
```

```
ans =  13.750
```

Se non mettiamo le parentesi, noi eseguiremo l'operazione $10+3^2 5/4$, che è ben diversa.

Facciamo 5/4 non dividiamo tutto il numeratore per 4 !!

```
» 10+3^2*5/4
ans =  21.250
```

2.5 L'OPERATORE DI ASSEGNAZIONE =

È importante sottolineare subito che il segno di uguaglianza che viene messo dopo una variabile prende il nome di operatore di assegnazione o di sostituzione a seconda che noi assegniamo per la prima volta ad una variabile una certa quantità numerica oppure stiamo cambiando il valore di quella variabile che è stata già definita in precedenza. Se scriviamo `a=3` (e la variabile `a` non è presente nel Workspace) allora noi stiamo assegnando lo scalare 3 alla variabile `a`. Se poi scriviamo `a=5`, stiamo modificando il valore di `a` che non conterrà più 3 ma 5. Possiamo anche fare altre operazioni come `a=a+4` oppure `a=2*a+1` e così via, utilizzando quindi il valore che la variabile contiene già. Nel primo caso, ad esempio, se `a=5`, la formula `a=a+4` aggiunge al valore di `a` 4 e il risultato lo assegna di nuovo alla variabile `a` che ora varrà 9 e non più 5.

```
» a=5
a =  5
» a=a+4
a =  9
```

Notiamo quindi che il simbolo di uguaglianza non ha lo stesso significato che ha in matematica per le equazioni. L'espressione `a=a+4` visto come un'equazione matematica ci darebbe `0=4`, un risultato non valido!!!

Diventa chiaro quindi che a sinistra del segno di uguaglianza possiamo mettere una sola variabile. Abbiamo errori se facciamo operazioni del tipo `5=a` oppure `a+4=a` o, ancora, `a+2=10`,... Octave ci dà il seguente messaggio

```
invalid constant left hand side of assignment
```

Se poi vogliamo assegnare ad una variabile il valore di un'altra variabile che non è stata tuttavia definita, abbiamo un messaggio di errore.

```
» a=5
a =  5
» a=a+t
error: 't' undefined near line 1 column 5
```

2.6 FUNZIONI MATEMATICHE

Abbiamo detto che possiamo usare inizialmente (per prendere mano) Octave come una calcolatrice. Vediamo allora le funzioni

di base, che usiamo abitualmente con la calcolatrice, come vanno usate in Octave.

Facciamo degli esempi.

```
» a=sin(20)
a =  0.91295
» a=exp(1)
a =  2.7183
» a=log10(2)
a =  0.30103
```

Abbiamo calcolato rispettivamente $\sin(20)$, e^1, $\log_{10} 2$. Al nome della funzione, facciamo seguire il valore in cui valutare la funzione scritto tra parentesi tonde.

!!Il valore della funzione sin viene dato in radianti!!!

Al posto dei numeri potremmo mettere anche una variabile.

```
» x=20
x =  20
» a=sqrt(x)
a =  4.4721
```

Abbiamo calcolato $\sqrt{x}$ con $x = 20$.

Vediamo quindi le principali funzioni (di uso più comune) che ci possono servire e che troviamo in una calcolatrice.

Le calcolatrici scientifiche ci permettono di fare tantissime cose: spulciando in rete tra i vari modelli, ce ne sono di quelli che hanno più di 100 funzioni fino ad arrivare ai modelli che hanno più di 500 funzioni. In tabella noi consideriamo le principali funzioni matematiche, quelle di cui faremo maggiore uso.

Se vogliamo conoscere altre funzioni matematiche, possiamo scrivere `doc` sulla Command Window e si aprirà una finestra con tutta la documentazione presente, tra cui quella corrispondente alle funzioni matematiche.

Funzione	Uso (con la variabile x)
e^x	`exp(x)`
$\ln(x)$	`log(x)`
$\log_{10}(x)$	`log10(x)`
$\cos(x)$	`cos(x)`
$\sin(x)$	`sin(x)`
$\tan(x)$	`tan(x)`
$\cos^{-1}(x)$	`acos(x)`
$\sin^{-1}(x)$	`asin(x)`
$\tan^{-1}(x)$	`atan(x)`
$x!$	`fact(x)`
x^{-1}	`inv(x)`
$\sqrt{x}$	`sqrt(x)`

Sottolineiamo ancora una volta che le funzioni trigonometriche che abbiamo visto in Tabella hanno come argomento il valore dell'angolo espresso in radianti. Noi lavoreremo sempre con queste funzioni trigonometriche.

Se invece si vuole lavorare in gradi, ci sono le funzioni `sind`, `cosd`, `tand`, `asind`, `acosd`, `atand`.

2.7 Costanti predefinite

Immaginiamo di dover lavorare con funzioni trigonometriche e di lavorare con la quantità π. Per avere in modo preciso il valore di π possiamo rifarci alle funzioni trigonometriche: sappiamo che $\pi = 2\sin^{-1}(1)$ oppure $\pi = 4\tan^{-1}(1)$. Perciò potremmo introdurre una variabile in questo modo e poi lavorare con essa:

```
» pigreco=2*asin(1)
pigreco =  3.1416
» a=sin(pigreco/4)+cos(pigreco/6)
a =  1.5731
```

In Octave, tuttavia, possiamo evitare di definire la variabile `pigreco` perchè esiste già una variabile predefinita, dal nome `pi` che è già il nostro π. Digitiamo `pi` sulla Command Window per vedere cosa abbiamo.

```
» pi
ans =  3.1416
```

Possiamo quindi usare `pi` ogni volta che ne abbiamo bisogno senza doverla definire.

Attenzione, però! ‼Se ci dimentichiamo che esiste già questa variabile e ne chiamiamo una noi in questo modo, allora noi andiamo a modificare la variabile `pi`.

```
» pi=100
pi =  100
» pi
pi =  100
```

⚠‼Quindi se si deve usare π non si scriva 3.14 all'interno del nostro programma. Purtroppo l'ho visto fare 😱‼. Sarà considerato un errore gravissimo (intanto perchè non ci ricordiamo che esiste la variabile predefinita `pi` e poi perchè scriviamo π con sole due cifre decimali corrette!!!!)

Abbiamo assegnato a `pi` il valore 100. Ora non vale più π. Se digitiamo semplicemente `pi` abbiamo infatti `pi=100`. Come fare per tornar al valore di `pi`=π senza dover chiudere e riavviare Octave? Semplicemente cancellando la variabile `pi`. In tal modo cancelliamo la variabile però se richiamiamo `pi` andiamo a richiamare la costante predefinita e quindi torniamo al nostro π. Per cancellare una o più variabili abbiamo l'istruzione **clear** seguito dal nome della variabile o dai nomi delle variabili da cancellare. Nel nostro esempio:

```
» clear pi
» pi
ans =  3.1416
```

Cancelliamo la variabile, poi la richiamiamo e vediamo che è ciò che vogliamo. Altre variabili predefinite sono `eps`, `Inf`, `NaN`: `eps` dà il valore della precisione di macchina, mentre `Inf` e `NaN` rappresentanto ∞ e il NotANumber.

! 🆘Ci sono altre variabili predefinite: `i`, `I`, `j`, `J` rappresentano tutte l'unità immaginaria $i = \sqrt{-1}$. Noi non useremo i numeri complessi e quindi useremo queste variabili con diverso significato. In Octave (ma solo in Octave) c'è anche `e`, con il significato del numero di Nepero $e = e^1$. Noi però non useremo questa variabile (se ne abbiamo bisogno scriveremo `exp(1)` in quanto MATLAB® non ha questa costante predefinita.)

2.8 ALTRI COMANDI UTILI

Abbiamo visto ora il comando **clear**. Se vogliamo cancellare una o più variabili presenti nel Workspace scriviamo **clear** seguito dal nome delle variabili, lasciando uno o più spazi bianchi tra una variabile e l'altra ma SENZA METTERE VIRGOLE.

```
» clear x pigreco
```

Se vogliamo cancellare tutte le variabili, basta scrivere **clear** e schiacciare il tasto di invio.

Il Workspace tornerà ad essere vuoto.

Un controllo sulle variabili presenti nel proprio spazio di lavoro può essere fatto anche attraverso altri comandi, quali **who** e **whos** Vediamo la differenza tra i due comandi:

```
» who
Variables in the current scope:

a    ans

» whos
Variables in the current scope:

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        a           1x1                          8  double
        ans         1x1                          8  double

Total is 2 elements using 16 bytes
```

L'istruzione **who** elenca le variabili presenti, mentre **whos** specifica anche il tipo e la dimensione occupata da ciascuna variabile (argomento che vedremo più avanti).

Altri comandi utili sono

- **clc**: agisce come una sorta di cancellino da lavagna; pulisce la Command Window e riporta il cursore >> in alto a sinistra.
- **help**: seguito dal nome di una funzione o di un comando spiega come utilizzarlo (ad esempio `help cos`, `help clc`).
- ;: se non vogliamo visualizzare il risultato di un'operazione di assegnazione, aggiungiamo un punto e virgola alla fine dell'istruzione. In questo modo, sulla stessa riga possiamo scrivere più istruzioni.

```
» a=3*2; b=5; c=a+b;
»
```

- Per visualizzare il contenuto di una variabile (lo abbiamo visto prima per `pi`) scriviamo il nome della variabile e schiacciamo il tasto di INVIO.

```
» a
a =  6
»
```

- Per visualizzare il contenuto di una variabile possiamo usare anche la funzione **disp**:

```
» disp(a)
  6
```

!!Si noti la differenza tra scrivere la variabile e usare la funzione **disp**.

- Se l'istruzione che dobbiamo scrivere è troppo lunga e vogliamo andare a capo, basta che scriviamo tre puntini ... per poter proseguire l'istruzione sulla riga successiva

```
» a= 5 + (10 +4 -3^2)/(23-12+4*8^3) - ...
30*12/(15/8)
a = -187.00
```

- Se vogliamo scrivere un'istruzione e poi commentarla, si mette il simbolo del percentuale % e tutto ciò che viene scritto dopo risulta un commento. Certo questa istruzione non la useremo mai sulla Command Window ma sarà utile quando scriveremo i nostri programmi. Tuttavia è bene prendere familiarità con essa già da ora.

2.9 SUGGERIMENTI DI TIPO PRATICO

Se dobbiamo scrivere delle istruzioni che abbiamo già scritto in precedenza sulla Command Window, al posto di riscrivere (tasto dopo tasto) tutte le lettere dell'istruzione, possiamo seguire due strade:

- usare la freccia ↑ che va in alto che si trova sulla tastiera del computer, in modo da andare indietro con i comandi già fatti;
- iniziare a scrivere la parola e poi, da tastiera, schiacciare due volte il tasto TAB (quello con la doppia freccia ↔) in modo da completare la parola più facilmente. Questa strada è utile anche se non ci ricordiamo il nome di una funzione che vogliamo utilizzare ma sappiamo che inizia in un certo modo: il tasto TAB ci aiuta a ricordare tutte le funzioni presenti che iniziano in quel modo.

```
» po
poisscdf    pol2cart    polyarea    polygcd     polyval     postpad
poissinv    polar       polyder     polyint     polyvalm    pow2
poisspdf    poly        polyeig     polyout     popen       power
poissrnd    polyaffine  polyfit     polyreduce  popen2      powerset
```

3

Tipi di dati

"Se ordinassi ad un generale di volare da un fiore all'altro come una farfalla, o di scrivere una tragedia, o di trasformarsi in un uccello marino; e se il generale non eseguisse l'ordine ricevuto, chi avrebbe torto, lui o io?"
"L'avreste voi", disse con fermezza il piccolo principe.
"Esatto. Bisogna esigere da ciascuno quello che ciascuno può dare", continuò il re.
Antoine de Saint-Exupèry

Negli esempi che abbiamo visto, i risultati numerici visualizzati sulla Comand Window presentano solo quattro cifre significative.

```
» sqrt(2)
ans =  1.4142
» pi
ans =  3.1416
```

Quattro cifre significative possono essere poca cosa se a noi interessano più cifre significative (negli esericizi di Calcolo Numerico, ad esempio, si devono riportare i risultati con ALMENO 7 CIFRE SIGNIFICATIVE 😯!)

Fatta l'importante precisazione che i calcoli in Octave vengono fatti tutti in doppia precisione (a meno che non si decida di cambiare tramite opportune funzioni), e visto che le variabili risultano in doppia precisione (si faccia **whos** o si veda nel Workspace: leggiamo `double` accanto al nome della variabile `ans` appena utilizzata), noi leggiamo quattro cifre significative dopo la virgola perchè la visualizzazione viene fatta, di default, in un formato con quattro cifre decimali! 😬

Possiamo cambiare il formato quando vogliamo: basta scrivere l'istruzione giusta 😬. Vediamo quali istruzioni possiamo usare e in che modo cambia il formato, tramite degli esempi.

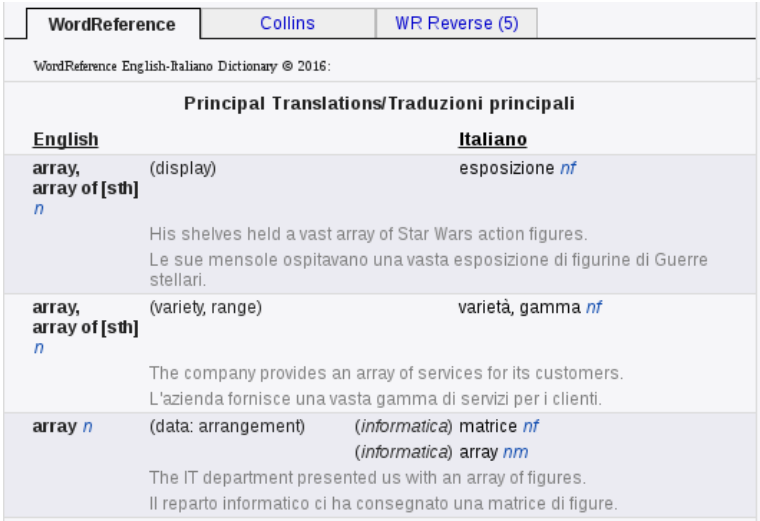
```
» a=sqrt(2)
a =  1.4142
» format long
» a
a =  1.41421356237310
» format short e
» a
a =    1.4142e+00
» format long e
» a
a =    1.41421356237310e+00
```

Vediamo un formato con 14 cifre decimali dopo il punto decimale, o un formato di tipo esponenziale. Se vogliamo tornare al formato di default possiamo scrivere `format` o `format short`. Ci sono altri tipi di formato che possono essere usati ma per conoscerli rimandiamo all'**help format**.

Osserviamo che non si usa la *virgola* per rappresentare il numero con le sue brave cifre decimali. Si usa (e si deve usare) il *punto decimale*. Se vogliamo scrivere dei numeri con la *virgola*, dobbiamo quindi scriverli con il punto decimale!

3.1 ARRAYS

Volevo cercare un termine in italiano che traducesse la parola *Array* ed ecco quello che ho trovato (faccio la foto al risultato della ricerca in rete). Come termine informatico *array* si traduce



WordReference | Collins | WR Reverse (5)

WordReference English-Italiano Dictionary © 2016:

Principal Translations/Traduzioni principali

English		Italiano
array, array of [sth] *n*	(display)	esposizione *nf*
	His shelves held a vast array of Star Wars action figures.	
	Le sue mensole ospitavano una vasta esposizione di figurine di Guerre stellari.	
array, array of [sth] *n*	(variety, range)	varietà, gamma *nf*
	The company provides an array of services for its customers.	
	L'azienda fornisce una vasta gamma di servizi per i clienti.	
array *n*	(data: arrangement)	(*informatica*) matrice *nf* (*informatica*) array *nm*
	The IT department presented us with an array of figures.	
	Il reparto informatico ci ha consegnato una matrice di figure.	

Figura 3.1: Fonte http://www.wordreference.com/enit/array

matrice o rimane array.

Noi diremo array perchè siamo interessati ad un aspetto di Octave che ci permetterà, d'ora in poi, di lavorare non più come se avessimo davanti una semplice calcolatrice ma un vero e proprio ambiente di programmazione.

Perchè tutta questa introduzione? Per focalizzare l'attenzione sul fatto che potremo lavorare, d'ora in poi, raggruppando insieme variabili che riguardano lo stesso *argomento*, mettendole insieme in array e lavorando con questi array come una singola entità.

Con gli esempi fatti fino ad ora, nel Workspace risultavano sempre variabili di dimensione 1×1. Facendo un **whos** le nostre variabili hanno `size` 1×1. Questo perchè anche le nostre semplici variabili scalari sono un caso particolare di array, un array ridotto ad un solo elemento.

Noi potremo lavorare con classi di array di diverso tipo:

- numerico
- di caratteri

- logico
- a celle
- a struttura
- di function handle

Vediamo ••in breve ciascuna di queste classi (ad eccezione delle function handle di cui parleremo più avanti). È chiaro che poi lavoreremo quasi sempre con array di tipo numerico 😉

3.2 VARIABILI DI TIPO NUMERICO

Le variabili scalari possono essere viste come un caso particolare di matrice 1×1. Ciò significa che in Octave noi possiamo lavorare con matrici anche di diversa dimensione. Con degli esempi vediamo come introdurre variabili matriciali.

```
» A=[1 2 3; 4 5 6; 7 8 9]
A =

   1   2   3
   4   5   6
   7   8   9
```

Per scrivere la matrice `A` abbiamo usato le parentesi quadre e scritto i valori di ciascuna riga lasciando uno (o più) spazi bianchi tra un valore e il successivo. Per andare alla riga successiva abbiamo usato il punto e virgola.

```
» A(1,2)
ans =  2
» A(2,3)
ans =  6
```

```
» A(:,1)
ans =

   1
   4
   7
```

```
» A(2,:)
ans =

   4   5   6
```

Per visualizzare un singolo elemento della matrice, abbiamo scritto il nome della variabile e tra parentesi tonde abbiamo scritto gli indici di riga e di colonna dell'elemento da visualizzare `A(1,2), A(2,3)`.

I due punti : rappresentano un operatore.

Per visualizzare gli elementi di una colonna abbiamo scritto il nome della matrice e, tra parentesi tonda, abbiamo messo i due punti e l'indice della colonna da visualizzare `A(:,1)`.

Per visualizzare gli elementi di una riga abbiamo scritto il nome della matrice e, tra parentesi tonde, l'indice della riga e poi i due punti `A(2,:)`.

La virgola non ha il significato del punto decimale! ⚠Ricordiamocelo sempre!

Per scrivere gli elementi della stessa riga possiamo anche usare la virgola che ha, quindi il significato di separare gli elementi della stessa riga!

```
» A=[1,2,3; 4,5,6;7,8,9]
A =
```

```
    1    2    3
    4    5    6
    7    8    9
```

L'operatore : ci permette di estrarre anche una sottomatrice. Vediamo con un esempio

```
» B=A(2:3,1:2)
B =

    4    5
    7    8
```

All'interno della matrice possiamo usare `end` per individuare l'ultima riga o colonna.

`end` va usato solo all'interno di una matrice o di un vettore altrimenti facciamo un errore.

```
» A(2:end, 1:end-1)
ans =

    4    5
    7    8
```

Nel Workspace risulta che la variabile `A` è di classe `double` e dimensione 3×3.

Se eseguiamo il comando **whos** A risulta che occupa 72 Bytes di memoria (contro gli 8 Bytes della variabile scalare `a` usata in precedenza).

Facciamo un altro esempio introducendo dei vettori (che sono matrici di una sola riga o di una sola colonna).

```
» x=[1.5 2.2 3.45]
x =

      1.5000    2.2000    3.4500

» y=[1; 2; 3]
y =

    1
    2
    3
```

Per scrivere il vettore riga abbiamo scritto gli elementi lasciando uno (o più) spazi bianchi tra un elemento e l'altro (oppure si mette la virgola tra un elemento e il successivo). Per scrivere il vettore colonna abbiamo messo un punto e virgola dopo ogni elemento. Abbiamo usato sempre le parentesi quadre.

Se guardiamo il Workspace, vediamo che queste due variabili sono sempre di classe `double`, e di dimensione rispettivamente 1×3 e 3×1. Facendo un **whos**, notiamo che occupano 24 Bytes di memoria.

È giunto il momento di analizzare la classe `double` e i Bytes occupati in memoria 😱.

🆘‼️Abbiamo scritto le componenti del vettore `x` con numeri con cifre decimali. Se non mettessimo il punto decimale ma la virgola avremmo:

```
x=[1,5 2,2 3,45]
x=
1 5 2 2 3 45
```

La virgola separa le cifre! Il vettore, in questo modo, non ha più le tre componenti che volevamo dare!

Vedremo altri modi per scrivere vettori righe e colonne. Questo è solo un primo approccio.

3.2.1 SUL TIPO DI VARIABILI NUMERICHE

Quando noi introduciamo una variabile o facciamo dei calcoli numerici, le variabili sono memorizzate in doppia precisione e i calcoli fatti in doppia precisione. Perciò la classe di default delle nostre variabili è `double`. Una variabile scalare in doppia precisione occupa 8 Bytes di memoria.

A volte, però, avere tutta questa precisione non serve (ad esempio se si sta lavorando con variabili che sono sempre intere, oppure se ci serve la singola precisione). Perciò ci sono delle funzioni che ci permettono di passare dalla doppia precisione alla singola precisione o a una memorizzazione del numero come numero intero a 8, 16, 32 o 64 bit, con o senza segno.

Perciò il vettore di tre componenti occupa $8 \times 3 = 24$ Bytes di memoria e la matrice 3×3 occupa $8 \times 9 = 72$ Bytes.

Classe (e funzione)	Valore Minimo	Valore Massimo	Bytes
`double`	-1.79e+308	1.70e+308	8
`single`	-3.40e+038	3.40e+038	4
`int8`	-128	127	1
`int16`	-32768	32767	2
`int32`	-2.14e+09	2.14+09	4
`int64`	-9.22e+18	9.22+e18	8
`uint8`	0	255	1
`uint16`	0	65535	2
`uint32`	0	4.29+09	4
`uint64`	0	1.84+19	8

Tabella 3.1: Memorizzazione dei numeri: `int8`, `int16`, `int32` servono per memorizzare numeri interi con il loro segno, `uint8`, `uint16`, `uint32`, `uint64` servono per memorizzare i numeri positivi senza il segno. Il valore minimo e il valore massimo indicano l'intervallo in cui possono essere rappresentati i numeri per ciascuna classe senza incorrere in *overflow* o *underflow*.

Vediamo degli esempi.

```
» a=sin(12)+23
a =  22.463
» b=single(a)
b =  22.463
» intero=12345
intero = -12345
» k=int16(intero)
k = -12345
» intpositivo=150
intpositivo =  150
» d=uint8(intpositivo)
d = 150
» whos a b intero k intpositivo d
Variables in the current scope:

   Attr Name            Size                     Bytes  Class
   ==== ====            ====                     =====  =====
        a               1x1                          8  double
        b               1x1                          4  single
        intero          1x1                          8  double
        k               1x1                          2  int16
```

```
        intpositivo      1x1                              8  double
        d                1x1                              1  uint8

Total is 6 elements using 31 bytes

»
```

Se poi vogliamo tornare a variabili in doppia precisione, basta applicare la funzione `double`.

Poter scegliere come memorizzare le variabili può essere utile quando si lavora con matrici di grosse dimensioni, per poter ridurre il costo di memorizzazione (naturalmente se può andare bene lavorare in singola precisione o passando alla memorizzazione di tipo intero! ‼).

3.3 VARIABILI DI TIPO CARATTERE

Array di tipo carattere sono array che contengono stringhe di caratteri. Facciamo subito un esempio semplice.

```
» stringa='ciao come va?'
stringa = ciao come va?
» whos stringa
Variables in the current scope:

   Attr Name          Size                     Bytes  Class
   ==== ====          ====                     =====  =====
        stringa       1x13                        13  char

Total is 13 elements using 13 bytes
» disp(stringa)
ciao come va?
```

Abbiamo scritto una frase tra apici, e questa frase diventa il contenuto della variabile `stringa`. Abbiamo riempito 13 spazi (considerando anche gli spazi vuoti tra una parola e la successiva) e la variabile occupa 13 Bytes di memoria ed è un array di 13 componenti. Volendo si può mostrare solo una parte delle sue componenti (come si vede dalle righe seguenti):

```
» stringa(1:4)
ans = ciao
» stringa(6:9)
ans = come
```

Useremo stringhe di caratteri là dove ci sarà bisogno di scrivere e visualizzare qualche messaggio, ma ci serviranno anche per fare grafici o visualizzare risultati numerici secondo un certo formato (tutte cose che vedremo via via che serviranno 🙂).

Con le stringhe di caratteri possiamo anche creare matrici di stringhe ma, dal momento che non le useremo, non ci dilunghiamo sull'argomento.

3.4 Variabili di tipo logico

Se scriviamo una proposizione logica, il risultato della proposizione può essere solo di due tipi: vero o falso.

In Octave il risultato di una proposizione logica assume valore 1 se la proposizione è vera, assume valore 0 se la proposizione è falsa.

Per scrivere una proposizione logica faremo uso degli operatori logici come $>$, $<$, $==$. Vediamo un esempio introducendo due variabili e scrivendo delle disuguaglianze: saranno vere o false e quindi avranno valore 1 o 0.

Sugli operatori logici spenderemo molte parole più avanti.Qui vediamo solo qualche esempio semplice confrontando delle variabili scalari.

```
» a=10; b=3;
» a>b
ans =  1
» p=a<b
p = 0
» p= a==b
p = 0
» a=b
a =  3
» p= a==b
p =  1
»
```

Le variabili `ans` o `p` risultano variabili logiche di dimensione 1×1.

Osserviamo che per confrontare se una variabile sia uguale (come contenuto) all'altra, abbiamo messo $==$ perchè l'operatore $=$ serve solo per assegnare il valore di una variabile o modificarlo!

3.5 Variabili di tipo cella o struttura

Con le variabili di tipo cella o struttura non lavoreremo molto (direi quasi che non lavoreremo affatto), perciò diamo solo una breve presentazione di questi due tipi di variabili.

Se vogliamo avere un array con stringhe di caratteri e tabelle di numeri possiamo farlo creando un array di celle.

Supponiamo di voler creare un array che ci dia qualcosa di simile al contenuto di questa tabella:

😇Saltate pure queste pagine, se non vi interessano. 😄Leggete, invece, se vi incuriosiscono ℹ️

Studenti per anno		Esame passato
158	2012	70
180	2013	82
172	2014	68
158	2015	75

I numeri scritti si riferiscono agli studenti di una imprecisata disciplina 😄

Dobbiamo mettere insieme stringhe di caratteri e matrici di numeri. Vediamo in che modo:

```
» A(1,1)={'Studenti per anno'};
» A(1,2)={'Esame passato'};
```

```
» A(2,1)={[158 2012; 180 2013; 172 2014; 158 2015]};
» A(2,2)={[70; 82; 68;75]}  %non mettiamo il punto e virgola per visualizzare
A =
{
  [1,1] = Studenti per anno
  [2,1] =

      158   2012
      180   2013
      172   2014
      158   2015

  [1,2] = Esame passato
  [2,2] =

     70
     82
     68
     75

}
»
```

Il comando **whos** A ci dice che la variabile A è 2×2, occupa 126 Bytes ed è fa parte della classe di tipo cell.

Osserviamo l'uso delle parentesi graffe per definire il contenuto di ciascuna cella.

Le parentesi graffe possono essere usate anche al posto delle parentesi tonde (scrivamo `A{1,1}='Studenti per anno'` al posto di `A(1,1)={'Studenti per anno'}`, spostando le parentesi graffe.). Vediamo un esempio con un altro array di celle.

```
» nome{1}='Giacomo';
» nome{2}='Ugo';
» nome{3}='Alessandro';
» nome
nome =
{
  [1,1] = Giacomo
  [1,2] = Ugo
  [1,3] = Alessandro
}
```

Un altro modo ancora è dato da quest'altro esempio

```
» altrinomi={'Beatrice','Laura','Fiammetta'};
» altrinomi
altrinomi =
{
  [1,1] = Beatrice
```

```
  [1,2] = Laura
  [1,3] = Fiammetta
}
```

Se non usassimo le parentesi graffe e mettessimo le quadre, avremmo delle stringhe di caratteri, con una gran differenza. Avremmo infatti qualcosa del genere

```
» ancoranomi=['Primo','Secondo','Terzo']
ancoranomi = PrimoSecondoTerzo
```

La variabile `ancoranomi` è una variabile di tipo carattere non di tipo cell (e unisce tutte le parole messe tra apici!).

Passiamo ora alle variabili di tipo struttura. Facciamo un esempio per capire di cosa si tratta. Pensiamo di voler scrivere un elenco in cui, per ciascuno studente, oltre a nome e cognome, vogliamo inserire il numero di matricola, l'indirizzo email e la città di provenienza. Possiamo fare qualcosa del genere costruendo la struttura che chiamiamo `studente` a cui associamo diversi campi:

```
» studente.nome='Pertinace';
» studente.matricola='123456789';
» studente.email='pertinace@studenti.unipd.it';
» studente.provenienza='Roma Antica';
» studente
studente =

  scalar structure containing the fields:

    nome = Pertinace
    matricola = 123456789
    email = pertinace@studenti.unipd.it
    provenienza = Roma Antica
```

Risulta che `studente` ha dimensione 1×1, occupa 56 Bytes ed è di Class struct (cioè structure).

Se vogliamo aggiungere altri studenti, basta che aggiungiamo tra parentesi il numero 2 (3, 4, ...) dopo il nome della variabile `studente` e prima dei campi che contraddistinguono la struttura, nel modo seguente.

```
» studente(2).nome='Flavia';
» studente(2).matricola='123456700';
» studente(2).email='flavia@studenti.unipd.it';
» studente(2).provenienza='Pompei';
» studente
studente =

  1x2 struct array containing the fields:

    nome
```

```
    matricola
    email
    provenienza

» studente(1)
ans =

  scalar structure containing the fields:

    nome = Pertinace
    matricola = 123456789
    email = pertinace@studenti.unipd.it
    provenienza = Roma Antica

» studente(2)
ans =

  scalar structure containing the fields:

    nome = Flavia
    matricola = 123456700
    email = flavia@studenti.unipd.it
    provenienza = Pompei
```

Carino ma purtroppo non lavoreremo con celle e strutture.
Però potrebbero tornare utili in futuro 😄

4

PROPOSIZIONI E PREDICATI LOGICI

Il bene si fa, ma non si dice. E certe medaglie si appendono all'anima, non alla giacca.
Gino Bartali

PER POTER IMPARARE a programmare, abbiamo ancora bisogno di alcuni strumenti di base. Dobbiamo essere in grado di scrivere delle proposizioni o dei predicati logici perchè molte istruzioni che faremo si basano sull'uso delle proposizioni logiche.

Semplici esempi li abbiamo già visti (`a>b`, `a==b`, ...) con predicati che possono essere o veri o falsi.

Approfondiamo, ora, questo argomento.

4.1 OPERATORI RELAZIONALI

Gli operatori relazionali sono i seguenti:

Tabella 4.1: Operatori relazionali

Operatore relazionale	Simbolo matematico	Significato
$<$	$<$	strettamente minore
$>$	$>$	strettamente maggiore
$<=$	$\leq$	minore o uguale
$>=$	$\geq$	maggiore o uguale
$==$	$=$	uguale
$\sim=$	$\neq$	non uguale

Il simbolo che vediamo in $\sim=$ si chiama *tilde*.

Come abbiamo già osservato in precedenza (ma *Repetita, iuvant!* 😉) il segno di uguaglianza è dato da due segni di uguaglianza $==$, perchè il segno $=$ è riservato come operatore di assegnazione. Gli altri simboli formati da due caratteri sono scritti senza lasciare spazi bianchi.

Questi operatori sono usati come operatori aritmetici all'interno di espressioni matematiche. Ci serviranno per le strutture di controllo (i cicli `if`, `while`, ...).

Quando sono messi a confronto due scalari, il risultato sarà 1 se la proposizione logica risulta vera, sarà invece 0 se la proposizione risulta falsa.

```
» a=1; b=100;
» a>=b
ans = 0
```

Se, invece, sono messi a confronto due array (della stessa dimensione **!!**), il confronto viene fatto elemento per elemento e quindi il risultato dà vita ad un array di tipo logico.

```
» a=[1 3 5]; b=[2 1 3];
» p=a<b
p =

   1   0   0
```

Il vettore `p` è una variabile logica di dimensione 1×3, e le sue componenti sono il risultato del confronto $a(1) < b(1)$? $a(2) < b(2)$? $a(3) < b(3)$? Nel primo caso la risposta è vera, e quindi abbiamo 1, negli altri due casi la risposta è falsa e abbiamo 0.

Le variabili logiche trovano un'interessante applicazione in problemi di questo tipo:

Problema

Dato un vettore, vogliamo prendere, del vettore, solo le componenti che sono minori di una certa quantità.

Il problema è del tutto simile se cambiamo l'operatore logico!

Sia, ad esempio, `x=[1 2 5 8 10 4 6 9 21 3 12 7 -1 -4 -6]` e vogliamo prendere solo le componenti che sono minori o uguali a 5.

In questo caso il vettore ha 15 componenti e non è cosi complicato risolvere manualmente il problema. Ma perchè farlo a mano se con poche istruzioni si può lasciar risolvere questo problema a Octave? Vediamo come

```
» x=[1 2 5 8 10 4 6 9 21 3 12 7 -1 -4 -6]
x =

    1    2    5    8   10    4    6    9   21    3   12    7   -1   -4   -6

» p=x<=5
p =

   1   1   1   0   0   1   0   0   0   1   0   0   1   1   1

» t=x(p)
t =

   1   2   5   4   3  -1  -4  -6
```

L'istruzione `x(p)` prende solo le componenti del vettore `x` in cui l'array logico `p` vale 1. Possiamo riassumere queste istruzioni in una sola, nel modo seguente

```
» t=x(x<=5)
t =

   1   2   5   4   3  -1  -4  -6
```

Il vettore logico è stato utilizzato, quindi, per estrarre elementi da un vettore e, in particolare, per estrarre le componenti corrispondenti al valore 1 del vettore logico.

Non potremmo fare la stessa cosa se il vettore avesse componenti 1 e 0 ma non fosse un vettore di tipo logico!

4.2 OPERATORI LOGICI

Gli operatori logici sono mostrati in Tabella 4.2.

Operatore logico	Significato
&	AND logico
|	OR logico
∼	NOT Negazione logica
& &	AND *short-circuit*
||	OR *short-circuit*

Tabella 4.2: Operatori logici

Questi operatori lavorano su proposizioni logiche. Dati `A` e `B` due array di tipo logico della stessa dimensione, avremo come risultato un array di tipo logico della stessa dimensione, le cui componenti sono il risultato dell'applicazione dell'operatore logico sugli elementi corrispondenti dei due array. Ad esempio, `P=A&B` avrà l'elemento P_{ij} (riga i e colonna j) uguale al risultato dato da $A_{ij}\&B_{ij}$.:

Osserviamo una differenza tra Octave e MATLAB® sugli operatori *short-circuit*. Octave fornisce un risultato anche se gli array logici da confrontare non sono scalari. Invece MATLAB® lavora solo su array scalari. E ciò che faremo anche noi.

- `A & B` dà come risultato un array i cui elementi valgono 1 dove i corrispondenti elementi di `A` e `B` sono entrambi uguali a 1, 0 se uno dei due corrispondenti elementi di `A` e `B` vale 0.
- `A | B` dà come risultato un array i cui elementi hanno valore 1 là dove almeno un elemento corrispondente in `A` o in `B` vale 1, invece ha valore 0 se entrambi i valori corrispondenti nei due array valgono 0.
- `∼ A` dà come risultato un array i cui valori valgono 0 se i corrispondenti valori di `A` valgono 1, viceversa valgono 1 se i corrispondenti valori di `A` valgono 0.
- `A & & B` restituisce un solo valore logico (un array logico scalare) !!Dà 1 solo se entrambi `A` e `B` valgono 1, 0 altrimenti.
- `A || B` restituisce un array logico scalare !!con valore 1 se una delle due variabili vale 1, 0 altrimenti.

Questi risultati si hanno considerato che 1 significa VERO e 0 significa FALSO. Se congiungiamo due proposizioni con la &, il risultato sarà vero se entrambe le proposizioni sono vere; il risultato sarà falso se anche solo una delle due proposizioni risulta falsa. Se usiamo la congiunzione | (OR, oppure, o) per mettere insieme due proposizioni, il risultato sarà vero se almeno una delle due proposizioni è vera; sarà falso se entrambe le proposizioni sono false. Vediamo degli esempi costruendo degli array logici, nel modo seguente: scriviamo delle matrici con elementi 1 o 0 e poi convertiamo la matrice in array logico tramite la funzione logical. Facciamo degli esempi con array di dimensione 2×2.

Pindemonte va a sciare d'inverno E va a mare d'estate sarà una frase vera se entrambe le proposizioni di cui è composta risultano vere. Se Pindemonte non va a sciare, la proposizione risulterà falsa. Se invece diciamo *Pindemonte va a sciare d'inverno O va a mare d'estate* risulterà vera anche se Pindemonte non va a sciare (quindi anche se la prima proposizione risulta falsa) a condizione che vada effettivamente a mare in estate!

La funzione logical converte gli elementi di un array in un array logico. Tutti i valori diversi da zero vengono convertiti nel valore VERO, cioè 1, i valori uguali a 0, vengono convertiti in FALSO, nello 0 logico.

```
=[1 1; 0 1]; A=logical(A)
A =

   1   1
   0   1

» B=logical([0 1; 1 0])
B =

   0   1
   1   0

» P=A&B
P =

   0   1
   0   0

» P=A|B
P =

   1   1
   1   1

» P=~A
P =

   0   0
   1   0
```

Vediamo ora degli esempi con gli operatori *short-circuit*.

```
» A=logical(1); B=logical(0);
» p=A&&B
p = 0
» p=A||B
p =  1
```

```
» B=A;
» p=A&&B
p =  1
```

Se l'elemento del primo array vale 0, e c'è la congiunzione &&, la risposta varrà 0 e quindi non ha senso andare a controllare il secondo array! Se stiamo usando la disgiunzione ||, se il primo array vale 1, la risposta sarà 1 qualsiasi sia il valore del secondo array (che non verrà valutato)! Perciò si parla di *short-circuit*. Ed è per questo che, lavorando tra array scalari, ha senso utilizzare sempre gli operatori *short-circuit*.

Consideriamo altri esempi

```
» a=10; amax=100; b=0.1; bmin=1.e-6;
»  p= a<=amax && b>bmin
p =  1
» a=101;
»  p= a<=amax && b>bmin
p = 0
» a=30; b=1.e-7;
»  p= a<=amax && b>bmin
p = 0
```

Per leggere meglio quanto stiamo scrivendo possiamo usare delle parentesi tonde in modo da isolare ciascuna proposizione:

```
»  p= (a<=amax) && (b>bmin)
```

5

Programmi, codici e pseudocodici

1. Le persone sono illogiche, irragionevoli ed egoiste. AMALE COMUNQUE.
2. Se fai del bene la gente ti accuserà di avere un secondo fine. FAI COMUNQUE DEL BENE.
Kent M. Keith

Iniziamo a vedere come usare Octave per programmare, partendo dal significato stesso di programma.

Se abbiamo un algoritmo, vale a dire una sequenza ordinata di istruzioni da eseguire per tradurre un certo problema, possiamo tradurre le istruzioni in un certo linguaggio di programmazione e scrivere, quindi, un programma da eseguire al calcolatore per poter risolvere numericamente il problema.

Un programma è dunque un insieme di istruzioni, scritte in un certo linguaggio, in modo da tradurre un algoritmo e permettere al calcolatore di risolvere un problema.

Noi focalizzeremo l'attenzione sui programmi da eseguire in ambiente Octave. I nostri programmi saranno compatibili con i programmi da eseguire in ambiente MATLAB® e li scriveremo in linguaggio MATLAB®, data la compatibilità che esiste tra linguaggio MATLAB® e linguaggio Octave.

Sottolineiamo una cosa importante che rende il linguaggio MATLAB® (Octave) molto più semplice da imparare rispetto ad altri linguaggi: un programma MATLAB® viene eseguito direttamente senza dover essere *compilato* come si fa con altri linguaggi di programmazione (come il FORTRAN). Inoltre le variabili che vengono utilizzate non vanno dichiarate all'inizio (cosa che si fa con altri linguaggi di programmazione). 😊

I file che scriveremo per i nostri programmi avranno l'estensione `.m` dove `m` sta per MATLAB®.

In particolare noi lavoreremo con due diversi tipi di programmi: **script** e `function` che corrispondono, più o meno, ai programmi e ai sottoprogrammi che abbiamo nominato a inizio dispensa.

5.1 Prime risposte alla domanda: perché script e function?

Qualcuno potrebbe chiedersi: perchè complicarsi la vita per imparare un linguaggio di programmazione e poi dover imparare a scrivere non un generico programma ma script e function, programmi e sottoprogrammi ??

Perchè non esiste un solo tipo di programma ?

E perchè conviene imparare a usare e gestire sia gli script che le function ❓

Perchè ❓

Rispondiamo in modo parziale a queste domande, partendo da un esempio. Approfondiremo poi l'argomento dopo aver capito bene cosa sono script e function.

🔄Vogliamo fare una media aritmetica dei voti presi agli esami del primo semestre. Le istruzioni che dobbiamo eseguire (usando una variabile per ciascun esame sostenuto) saranno:

- assegnare il voto preso a ciascun esame del primo semestre (`Analisi1=...; Economia= ....; ....`
- applicare la formula per calcolare la media dei voti (`Media=....`)
- stampare il risultato.

Le istruzioni che abbiamo scritto riguardano i dati di ingresso (input) del problema, l'algoritmo numerico (ottenere la media aritmetica dei dati di input), la stampa del risultato.

Risolveremo questo problema usando uno script.

Arriva la fine del secondo semestre e vogliamo aggiornare la media dei voti. Cosa facciamo? Possiamo aggiungere altri dati di input, e riscrivere la formula per ottenere la media dei voti. Quindi facciamo delle modifiche al nostro script!

Nello stesso tempo, ci viene proposto un altro problema: fare un'indagine nella cerchia dei nostri amici/conoscenti per capire quanti libri leggono in media i ragazzi che frequentano l'università. Dobbiamo scrivere un altro programma inserendo come variabili i libri letti dai nostri amici/conoscenti (e immaginiamo di avere, questa volta, 30 variabili), e poi fare la media. A questo punto, pensiamo: ma... 🤔avevamo già fatto un programma simile, dovevamo sempre calcolare la media (anche se con meno variabili)... possiamo sfruttare il lavoro già fatto ❓🙏

E a questo punto entrano in gioco le function (per questo esempio semplice, una function)! Se le istruzioni che corrispondono al calcolo della media aritmetica dei voti ottenuti nel primo semestre non fossero scritte all'interno dello script ma usando una function, allora quella stessa function potrebbe essere usata sia per aggiornare la media dei voti alla fine del secondo semestre, sia per calcolare il numero dei libri letti in media nella nostra cerchia di amici/conoscenti. Il nostro lavoro sarebbe quindi semplificato. Basterebbe dare i dati di ingresso del problema, utilizzare la function, ottenere il risultato (e questo per più problemi simili: e se pensiamo ad algoritmi che hanno tante istruzioni, vediamo come sia semplificato il lavoro da fare perché dobbiamo scrivere queste istruzioni solo una volta e non in ogni script!).

Passiamo quindi a vedere come scrivere script e function, partendo dagli script.

Teniamo presente che stiamo risolvendo un semplice problema in cui l'algoritmo sarà tradotto con un'istruzione di una riga. Se immaginiamo problemi più complicati in cui l'algoritmo ha bisogno di tante istruzioni da eseguire, avremo, ovviamente, programmi più complicati...

5.2 SCRIPT

Uno script traduce un algoritmo, risolvendo un problema con assegnati dati di input, utilizzando istruzioni di tipo sequenziale o mediante cicli di controllo. Viene eseguito sulla Command Window e, se si vuole risolvere lo stesso problema con dati di input diversi, occorre cambiarli (all'interno dello script – modificando quindi alcune istruzioni già scritte – o durante l'esecuzione dello script stesso, a seconda di come è stato strutturato il programma).

Per eseguire uno script dal nome (ad esempio) `script.m` si può scrivere, sulla Command Window, `script` (il nome del file senza il .m) e schiacciare il tasto di INVIO. Le istruzioni che abbiamo scritto nel nostro file verranno *capite* dal nostro ambiente di programmazione ed eseguite.

Riprendiamo l'esempio della media dei voti del primo semestre, per scrivere il nostro primo script.

Supponiamo che nel primo semestre abbiamo avuto questi voti: Analisi matematica 1 24, Elementi di chimica 28, Economia ed organizzazione aziendale 27.

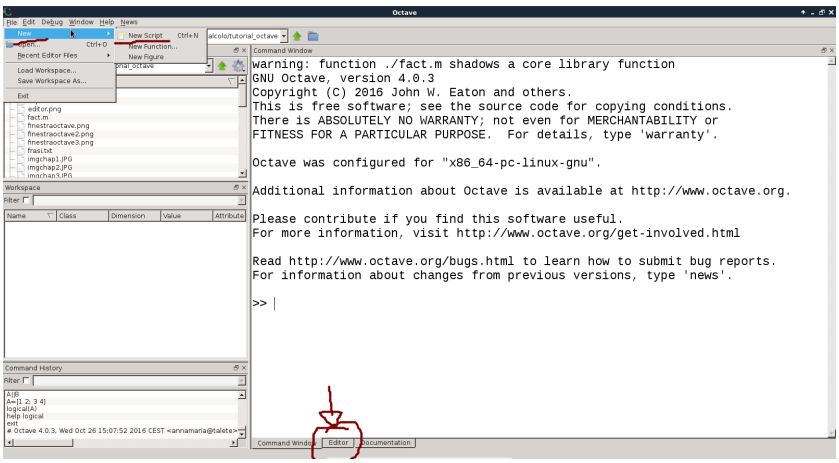
Dalla Command Window, per calcolare la media dei voti ottenuti faremmo semplicemente

```
» (24+28+27)/3
ans =  26.333
»
```

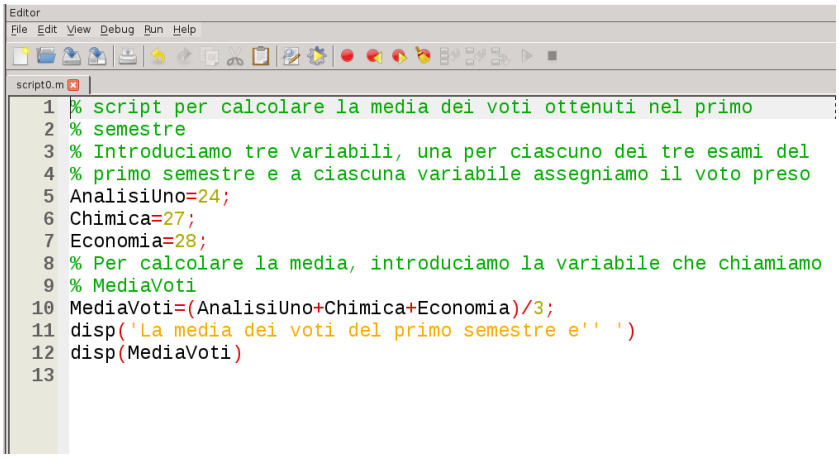
Applicheremmo, cioè, la formula della media (l'algoritmo, così semplice) e subito sapremmo la media dei voti. Una volta però chiusa la sessione di Octave, le istruzioni date verranno perse. Se quelle istruzioni invece vengono salvate in un file, allora possiamo rieseguirle quante volte vogliamo senza doverle riscrivere ogni volta! Piuttosto che mettere il valore numerico dei voti ottenuti, assegneremo questi voti a delle variabili. In questo modo possiamo facilmente modificarne il valore (perciò, se abbiamo già fatto due esami e ci manca il terzo, possiamo vedere subito come cambia la media se al terzo esame prendiamo 25 piuttosto che 28 o 30: basterà eseguire lo script cambiando il valore della variabile associata all'esame da fare). Inoltre, avendo delle variabili, capiremo meglio ciò che stiamo facendo. Lo script sarà un file di testo da salvare con un nome. A questo script diamo il nome `script0.m`. Per scriverlo usiamo l'editor di testo presente già in ambiente Octave seguendo una delle seguenti modalità (tutte equivalenti fra loro):

- dalla Command Window scriviamo `edit script0.m`
- sulla finestra di Octave, in alto a sinistra, clicchiamo su FILE, poi su NEW e infine su NEW SCRIPT (si veda anche la figura 5.2);

- in basso alla finestra di Octave, a destra della scritta Command Window, clicchiamo su EDITOR ed entriamo direttamente nell'editor di testo dove andare a scrivere le nostre istruzioni.



Nelle ultime due modalità, dopo aver scritto il file, lo salveremo dando il nome `script0.m`. Perciò dopo averlo scritto, troveremo il file `script0.m` nella finestra del File Browser.




```
% script per calcolare la media dei voti ottenuti nel primo
% semestre
% Introduciamo tre variabili, una per ciascuno dei tre esami del
% primo semestre e a ciascuna variabile assegniamo il voto preso
AnalisiUno=24;
Chimica=27;
Economia=28;
% Per calcolare la media, introduciamo la variabile che chiamiamo
% MediaVoti
MediaVoti=(AnalisiUno+Chimica+Economia)/3;
disp('La media dei voti del primo semestre e'' ')
disp(MediaVoti)
```

Questo script si compone di diverse parti:

- I commenti allo script (ciò che segue il simbolo % rappresenta un commento).
- I dati di input del problema (le variabili con il loro valore assegnato).
- L'algoritmo numerico (la media dei voti)
- La visualizzazione del risultato (tramite la function **disp**: sull'uso di questa function spenderemo due parole più avanti).

Lo script è molto semplice. Se vogliamo cambiare i dati, basta correggere i valori numerici alle variabili degli esami corrispondenti.

In questo script abbiamo riportato delle istruzioni che avremmo potuto scrivere sulla Command Window. Messe in uno script, abbiamo il vantaggio di poter rieseguire queste istruzioni quando vogliamo, senza dover riscrivere tutte le istruzioni.

Vediamo ora altri esempi che ci permettono di capire meglio in che modo tradurre un algoritmo in istruzioni da eseguire in Octave

Tutto ciò ci sarà utile per scrivere non solo script ma anche function.

!!La media dei voti si può fare anche usando una function già predefinita, che si chiama **mean**. Non l'abbiamo usata perchè stiamo facendo un esempio semplice per capire le differenze tra script e function.

5.3 PSEUDOCODICI

Per scrivere al meglio i nostri codici, conviene prima di tutto tradurre l'algoritmo in un diagramma di flusso (come abbiamo visto all'inizio di queste dispense) oppure scrivere uno pseudocodice, nel quale linguaggio naturale e espressioni matematiche sono utilizzate per costruire le istruzioni simili a quelle che dovremo dare al computer per eseguire il codice.

Uno pseudocodice ci aiuta a scrivere e commentare un codice.

Vediamo ora alcuni pseudocodici e traduciamoli in script in modo da imparare a programmare e a usare le diverse strutture.

5.3.1 DATI DI INPUT, ISTRUZIONI SEQUENZIALI, DATI DI OUTPUT

Problema: Si calcoli il perimetro p e l'area A di un triangolo, di cui sono assegnati i valori di due lati (a e b) e il valore dell'angolo γ (dato in radianti) da essi compresi.

Si calcoli la lunghezza del terzo lato c applicando la formula $c^2 = a^2 + b^2 - 2ab\cos\gamma$. Per calcolare l'area del triangolo si utilizzi la formula $A = \sqrt{s(s-a)(s-b)(s-c)}$, dove $s = p/2$ è il semiperimetro.

Soluzione con uno pseudocodice

1. Dare i dati di input dei valori dei due lati a e b e dell'angolo γ,

2. Calcolare il lato c.

$$c = \sqrt{a^2 + b^2 - 2ab\cos\gamma}$$

3. Calcolare il perimetro p.

$$p = a + b + c$$

4. Calcolare il semiperimentro s.

$$s = \frac{p}{2}$$

5. Calcolare l'area A.

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

6. Visualizzare i risultati di p e A.

Soluzione con uno script Per tradurre in linguaggio di programmazione il nostro pseudocodice, passiamo a scrivere il nostro secondo script, che salveremo con il nome `script1.m`.

Si tratta di uno script più serio rispetto al precedente e quindi passiamo dal livello 0 al livello 1 (script0.m il precedente script, scrip1.m questo).

```
% script1 per calcolare perimetro e area di un triangolo di cui sono
% dati due lati, a e b, e l'angolo (in radianti) da essi compreso
a=input('scrivi il valore del lato a: ');
b=input('scrivi il valore del lato b: ');
gam=input('scrivi il valore in radianti dell''angolo gamma :');
c= sqrt(a^2+b^2-2*a*b*cos(gam));
p=a+b+c;
s=p/2;
A=sqrt(s*(s-a)*(s-b)*(s-c));
disp('il perimetro vale ')
disp(p)
disp('l''area vale ')
disp(A)
```

```
% script1 per calcolare perimetro e area di un triangolo di cui sono
% dati due lati, a e b, e l'angolo (in radianti) da essi compreso
a=input('scrivi il valore del lato a: ');
b=input('scrivi il valore del lato b: ');
gam=input('scrivi il valore in radianti dell''angolo gamma :');
c= sqrt(a^2+b^2-2*a*b*cos(gam));
p=a+b+c;
s=p/2;
A=sqrt(s*(s-a)*(s-b)*(s-c));
disp('il perimetro vale ')
disp(p)
disp('l''area vale ')
disp(A)
```

Vediamo passo passo le istruzioni che abbiamo scritto:

- Le prime righe presentano un commento (c'è il %, quindi quello che segue è un commento: spieghiamo cosa fa lo script).

- le istruzioni a= **input**('.....') hanno come effetto quello che durante l'esecuzione dello script viene visualizzato sulla Command Window il contenuto della stringa di caratteri presenti tra apici all'interno di **input**. Il cursore nella Command Window rimarrà a lato della stringa, aspettando la nostra risposta. Ciò che scriveremo sarà assegnato alla variabile il cui nome è dato prima del segno di uguaglianza a **input**. Proviamo a scrivere una di queste istruzioni sulla Command Window, senza il punto e virgola:

```
» a=input('scrivi il valore del lato a: ')
```

Quando schiacciamo il tasto di invio avremo

```
» a=input('scrivi il valore del lato a: ')
scrivi il valore del lato a:
```

Se noi scriviamo un numero, per esempio 5, avremo

```
» a=input('scrivi il valore del lato a: ')
scrivi il valore del lato a: 5
a =  5
```

- Una volta dati i valori di input delle nostre variabili, applichiamo in modo sequenziale le istruzioni che ci permettono di ottenere il lato c, il perimetro, il semiperimetro e l'area.
- Con la funzione **disp** visualizziamo i risultati: si osservi che se scriviamo **disp**('il␣perimetro␣vale') noi visualizziamo una stringa di caratteri; se scriviamo **disp**(p) (non ci sono gli apici) visualizziamo il contenuto della variabile p.

Osserviamo che questo script richiede ogni volta che i dati di input siano dati da tastiera: il vantaggio è che possiamo cambiare i dati ogni volta che vogliamo, lo svantaggio è che, se vogliamo rieseguire lo script con gli stessi dati, dobbiamo comunque scriverli ogni volta da tastiera.

IMPORTANTE: se la stringa ha un apostrofo o un accento, noi mettiamo due volte di seguito l'apice all'interno della stringa! Ad esempio l'istruzione
disp('il␣valore␣dell''area␣e''␣:')
produce il risultato
`il valore dell'area e' :`

5.4 Istruzioni NON sequenziali

Iniziamo a complicare un po' le cose, pensando a istruzioni non sequenziali.

Riprendiamo le varie strutture viste all'inizio di questo tutorial e scriviamo, per ciascun esempio già visto, uno pseudocodice e uno script.

5.4.1 Ciclo if

Dobbiamo calcolare la radice quadrata di un numero dato in input. Rivedendo il diagramma di flusso, lo pseudocodice è:

1. Dare in input il valore x di cui calcolare la radice quadrata.
2. Se $x \geq 0$ allora si calcoli $y = \sqrt{x}$ altrimenti non calcolare la radice quadrata.
3. Visualizzare y o il messaggio che non si può calcolare.

Chiamiamo lo script con il nome `scriptradquadr.m`. Lo scriviamo in questo modo:

```
% script sull'uso del ciclo if
x=input('valore x di cui calcolare la radice quadrata ');
if x>=0
   y=sqrt(x);
   disp('la radice quadrata vale ')
   disp(y)
else
   disp('il valore di x non va bene');
end
```

Di questo script va vista subito la struttura del cosiddetto ciclo `if`.

Se (`If`) è vera la proposizione espressa da `x >=0` si può calcolare la sua radice quadrata, altrimenti (`else`) si visualizza il messaggio in cui si dice che il valore dato per x non va bene. Osserviamo come all'interno del ciclo if, a seconda che si calcoli o meno la radice quadrata di x, si produce anche l'output, cioè visualizziamo il risultato y oppure il messaggio.

Se non vogliamo visualizzare alcun messaggio per $x < 0$ potremmo ridurrre il ciclo if in questo modo:

```
if x>=0
   y=sqrt(x);
   disp('la radice quadrata vale ')
   disp(y)
end
```

Se è vera la proposizione logica espressa dopo if, fai una certa cosa. Se non è vera, non facciamo nulla.

Arricchiamo ora il ciclo if: se $0 < x < 4$ allora $y = \sqrt{x}$, se $4 \leq x < 6$, $y = 6 - x$, altrimenti (cioè se $x \leq 0$ oppure $x \geq 6$, non sono vere le condizioni precedenti) $y = 0$.

Vediamo come tradurre queste istruzioni:

```
if  0 < x && x < 4    % possiamo anche scrivere if (0<x) && (x<4)
                      % usando le parentesi tonde
   y=sqrt(x);
elseif  4 <= x && x <6
   y=6-x;
else
   y=0;
end
```

Osserviamo [1] che `elseif` va scritto tutto attaccato perchè fa parte di un'unica struttura condizionale. Infatti abbiamo un solo `end` finale.

[1] Se abbiamo dei cicli if all'interno di un ciclo if, allora scriveremo `else if` (staccati) e questo introdurrà un altro ciclo if (che dovrò poi essere chiuso da end).

5.4.2 CICLO FOR

Per capire il ciclo for, prendiamo l'esempio del poeta che deve scrivere 100 volte alla sua amata la stessa frase d'amore....

Per visualizzare i risultati, noi la scriveremo 10 volte.... (sarà contenta lo stesso 💕).

```
% script sull'uso del ciclo for
for i=1:10
  disp('You are my queen')
end
```

Eseguiamo:

```
» scriptcicloforpoeta
You are my queen
You are my queen
You are my queen
You are my queen
You are my queen
You are my queen
You are my queen
You are my queen
You are my queen
You are my queen
```

La variabile `i` usata all'interno del ciclo for, parte da 1 e viene incrementata di 1 fino ad arrivare a 10. Se ci facciamo stampare *i* ce ne rendiamo conto ancora meglio 👁.

```
for i=1:10
  disp(i)
  disp('You are my queen')
end
```

```
» scriptcicloforpoeta
 1
You are my queen
 2
You are my queen
 3
You are my queen
 4
You are my queen
 5
You are my queen
 6
You are my queen
 7
You are my queen
 8
You are my queen
```

```
 9
You are my queen
 10
You are my queen
```

Nel ciclo for possiamo usare una variabile di appoggio i che varia come nell'esempio, oppure con un certo incremento. Ad esempio, se vogliamo fare la somma dei primi n numeri interi dispari $1+3+5+7+9+...$ possiamo usare un ciclo for facendo partire i da 1 e usando un incremento uguale a 2, fino ad arrivare a $2n-1$. Abbiamo la seguente regola sulla variabile usata all'interno del ciclo for

`for var = variniziale :  incremento :  varfinale`

Si parte da `var=variniziale` e si va avanti con `var= var+incremento` fino ad arrivare a varfinale o al valore ad esso più vicino (per difetto). I valori dati a `variniziale, incremento` e `varfinale` possono essere numeri interi o reali. Si può dare un incremento negativo e, partendo da `variniziale` maggiore di `varfinale`, ottenere dei valori decrescenti per `var`.

Tornando alla somma dei primi n numeri dispari, in termini di pseudocodice, dobbiamo:

1. dare in input il valore n che indica quanti numeri dispari dobbiamo sommare tra loro, a partire dal numero 1
2. inizializzare a zero la variabile `somma`
3. sommare i primi n numeri dispari, all'interno di un ciclo che incrementa i numeri dispari.

```
disp('si␣vuole␣fare␣la␣somma␣dei␣primi␣n␣numeri␣dispari␣')
n=input('con␣quale␣n␣?␣')
somma=0;
for  i=1:2:2*n-1
  somma=somma+i;
end
disp('la␣somma␣e''␣:␣')
disp(somma)
```

È importante notare come la variabile in cui avremo il risultato finale (`somma`) sia stata inizializzata a zero prima del ciclo for. Questo ci serve perchè, all'interno del ciclo noi sommiamo ogni volta a `somma` il valore della variabile `i`. Perciò partendo da `somma=0`, avremo

1. `somma=somma+i= 0 +1 =1`
2. `somma=somma+i= 1 +3 =4`
3. `somma=somma+i= 4 +5 =9`
4. `somma=somma+i= 9 +7 =16`
5. `somma=somma+i= 16 +9 =25`

6. ...

Proviamo ora a moltiplicare tra loro i primi n numeri pari, a partire da 2. Il risultato lo salviamo nella variabile `prodotto`. Questa volta, useremo sempre un ciclo for e aggiorneremo di volta in volta la variabile `prodotto` che deve essere uguale a $2 \cdot 4 \cdot 6 \cdot 8 \cdot ....$ La variabile `prodotto` sarà inizializzata a 1, questa volta.

```
disp('si vuole fare il prodotto dei primi n numeri pari ')
n=input('con quale n? ')
prodotto=1;
for npari=2:2:2*n
  prodotto=prodotto*npari;
end
disp('il prodotto vale : ')
disp(prodotto)
```

Osserviamo che la variabile usata nel ciclo for l'abbiamo chiamata `npari`. Non ci sono restrizioni sul nome da dare.

Il ciclo for può essere utilizzato anche facendo variare la variabile all'interno di un vettore. Con un esempio capiremo meglio:

```
x=[1.5 -1.2 1.3 2.4];
for var=x
% scriviamo ora var senza il punto e virgola in modo da visualizzare
% la variabile var e capire meglio questo ciclo for
  var

end
```

Se lo eseguiamo, abbiamo

```
var =  1.5000
var = -1.2000
var =  1.3000
var =  2.4000
```

5.4.3 Ciclo while

Per capire come funziona il ciclo while, traduciamo in script il diagramma di flusso dell'esempio che avevamo visto sul conto in banca... (andiamo a rivederlo nel primo capitolo).

```
C=1000;
anni=0;
while C< 20000
anni=anni+1;
C=C*1.02+3000;
end
disp(C)
disp(anni)
```

Abbiamo scritto:

`while proposizione logica`
esegui determinate istruzioni
`end`.

Notiamo che la proposizione logica non può essere *statica* (cioè non può non dipendere dalle istruzioni da eseguire all'interno del ciclo stesso) altrimenti dal ciclo non si esce mai e si entra in un *loop infinito*.

Nello script che abbiamo ora presentato, la proposizione logica è `C < 20000`, e all'interno del ciclo `C` viene aggiornato!

Il rischio di entrare in un *loop infinito* può essere sempre alle porte ‼️😉. Perciò è importante scrivere una proposizione logica che, ad un certo punto, non potrà essere vera! Il ciclo while infatti esegue le istruzioni all'interno del ciclo se la proposizione logica messa dopo la parola `while` è vera. Arrivati all'`end`, torna indietro dove c'è la proposizione logica e se è vera esegue le istruzioni del ciclo. Si esce dal ciclo quando la proposizione logica non è più vera! Quindi le istruzioni del ciclo vengono iterate più e più volte, fino a quando la proposizione logica risulta falsa.

Vediamo degli esempi.

```
x=2;
while x<= 30
   x=2*x+1
end
```

Quando eseguiamo abbiamo:

```
x =  5
x =  11
x =  23
x =  47
```

Cambiamo ora il valore `x` iniziale.

```
x=0.1;
while x <=30
   x=x.^2
end
```

Quando eseguiamo, il ciclo non ha mai fine perchè i valori diventano sempre più piccoli, quindi `x<=30` vale sempre ‼️. Per interrompere l'esecuzione del ciclo digitiamo i tasti CONTROL e C contemporeaneamente!

Per evitare questa brusca interruzione conviene inserire un contatore che ci permetta di entrare nel ciclo per un certo numero massimo di volte. Vediamo come.

```
x=0.1;
contatore=0;
while (x <=30) && (contatore<100)
   contatore=contatore+1;
   x=x.^2;
end
disp('contatore')
disp(contatore)
disp('valore x')
disp(x)
```

```
contatore
 100
valore x
0
```

In questo caso, la variabile `contatore` arriva a 100, quindi la proposizione `contatore <100` diventa falsa e l'intera proposizione risulta falsa (anche se `x <=30` rimane vera). Perciò si esce dal ciclo while.

5.4.4 CICLO SWITCH

Concludiamo l'elenco delle strutture non sequenziali con il ciclo `switch` che permette di fare una scelta e, in base alla scelta fatta, vengono eseguite determinate istruzioni (il ciclo è dunque simile al ciclo if). Vediamo come funziona partendo da un esempio.

Vogliamo convertire la misura di una temperatura, passando da Celsius a Fahrenheit, o da Kelvin a Celsius e così via. Nel nostro script diamo in input il valore della temperatura, l'unità della temperatura in cui la stiamo dando, e l'unità della temperatura in cui vogliamo convertirla. Come unità di temperatura, usiamo le più diffuse (Celsius, Fahrenheit e Kelvin)

```
% script sull'uso del ciclo switch
% per convertire il valore della temperatura da un'unita' di
% temperatura ad un'altra
clear
T=input('scrivi il valore della temperatura ')
disp('sia C per Celsius, F per Fahrenheit, K per Kelvin')
Uinp=input('scrivi l''unita'' di temperatura usata (C, F, K)', 's');
Uout=input('scrivi in quale unita'' trasformarla (C, F, K) ', 's');
% trasformiano la temperatura data in unita' Kelvin
% in modo poi da passare da Kelvin all'unita' richiesta
errore=0;
switch Uinp
  case 'C'
    TT= T+273.15;
  case 'F'
    TT=(T+459.67)/1.8;
  case 'K'
    TT=T;
  otherwise
    errore=1;
end
switch Uout
   case 'C'
     Tout=TT-273.15;
   case 'F'
     Tout=TT*1.8-459.67;
   case 'K'
     Tout=TT;
   otherwise
```

```
    errore=1;
end
if errore   %la proposizione errore e' vera se vale 1, cioe' se
             % e' stato commesso un errore (nel nostro esempio
             % se non sono stati messi correttamente i dati
             % per le unita' di misura).
 disp('non hai messo bene i dati di input ')
else
disp('il valore richiesto e'' ')
disp(Tout)
end
```

Commentiamo lo script:

- Osserviamo che usiamo la function **input** usando due stringhe: una in cui scriviamo ció che vogliamo sia messo in input, e un'altra in cui viene scritto semplicemente `'s'`. Ciò comporta che ciò che scriveremo (e che non sará un numero ma una stringa di caratteri, nell'esempio una sola lettera) verrà salvato in una variabile di tipo carattere, cioè una stringa di caratteri.

- quando scriviamo **switch** Uinp, viene letto il valore della variabile `Uinp` e, a seconda che valga `C, F` o `K` si andranno a eseguire, rispettivamente, le istruzioni dopo **case**'C', **case**'F' o **case**'K'. A seconda del caso si eseguono determinate istruzioni (in questo esempio, si scrive il valore della temperatura in gradi Kelvin partendo dal valore della temperatura nell'unità di misura data in input).

- C'è poi un secondo ciclo switch, del tutto simile al primo, che permette di passare dal valore della temperatura in Kelvin al valore della temperatura nell'unità di misura scelta a inizio programma.

È importante notare che nell'inserire i dati di input è facile sbagliarsi (nel senso che si può digitare `c` al posto di `C` oppure si scrive, senza volerlo, un'altra lettera. In tal caso il problema non può essere risolto correttamente e può quindi dare un messaggio di errore. Per risolvere elegantemente questo passaggio 🤭, introduciamo, prima del primo ciclo switch una variabile, che chiamiamo `errore` e che poniamo uguale a zero. All'interno di ogni ciclo switch, oltre a scrivere `case ...`, `case ...` , ... scriviamo `otherwise`: Se la variabile dello switch non corrisponde a nessuno dei casi indicati, allora vengono eseguite le istruzioni che seguono `otherwise`, ponendo la variabile `errore` uguale a 1. Se `errore=1` vuol dire che abbiamo assegnato un valore sbagliato alla variabile che viene usata nello switch. Perciò, poi usiamo un ciclo if: **if** errore (e non c'è bisogno di scrivere la proposizione `errore==1` perchè se `errore=1` la proposizione `errore` vale già 1 quindi è considerata vera) allora scriviamo un messaggio di errore, altrimenti visualizziamo il risultato di output del problema.

5.4.5 STRUTTURA DEI CICLI

Riassumiamo le strutture dei cicli che abbiamo studiato fino ad ora.

5.4.6 CICLO IF

```
....
.... % istruzioni
....
if proposizione che esprime una condizione
....
....  % primo gruppo di istruzioni da eseguire
....
elseif proposizione con condizione
....
....  % secondo gruppo di istruzioni da eseguire
....
else
....
....  % terzo gruppo di istruzioni da eseguire
....
end
....
.... % istruzioni
....
```

Questa struttura può essere arricchita di altre condizioni, aggiungendo degli **elseif** o può essere ridotta.

```
if  proposizione con condizione
....
....  % primo gruppo di istruzioni da eseguire
....
else
....
....  % secondo gruppo di istruzioni da eseguire
....
end
```

... e può essere ridotta ancora...

```
if  proposizione con condizione
....
....  % primo gruppo di istruzioni da eseguire
....
end
```

5.4.7 CICLO FOR

```
for indice= valiniziale:incremento:valfinale
....
```

```
....  % gruppo di istruzioni da eseguire
....
end
```

In questa struttura `indice` è la variabile del ciclo, `valiniziale` è il valore assunto al primo passo del ciclo da `indice`; `incremento` è l'incremento di `indice` ad ogni passo; `valfinale` è l'ultimo valore di `indice` (oppure `indice` arriva al più grande valore possibile minore di `valfinale` (riprendere gli esempi già visti !!).

All'interno del ciclo non si dovrebbe ridefinire la variabile `indice` (non ha senso farlo !!).

Oppure il ciclo for viene usato nel modo seguente

```
for indice= vettore
....
....  % gruppo di istruzioni da eseguire
....
end
```

Questa volta la variabile `indice` assume di volta in volta le componenti del vettore `vettore`.

5.4.8 CICLO WHILE

```
while proposizione logica
....
....  % gruppo di istruzioni da eseguire
....
end
```

Fintantochè risulta vera la proposizione logica verranno eseguite le istruzioni all'interno del ciclo.

5.4.9 CICLO SWITCH

```
switch   espressione dello switch
    case   valore1
        ....
        ....  % primo gruppo di istruzioni da eseguire
    case   valore2
        ....
        ....  % secondo  gruppo di istruzioni da eseguire
    case   valore3
        ....
        ....  % terzo gruppo di istruzioni da eseguire
    otherwise
        ....
        ....  % quarto gruppo di istruzioni da eseguire
end
```

5.5 IMPORTANTE A SAPERSI!

5.5.1 CICLI DENTRO CICLI

All'interno di una struttura possono essere usate altre strutture: per esempio all'interno di un ciclo for si può inserire un altro ciclo for oppure un ciclo if.... L'importante è che i diversi cicli siano scritti in modo che, quando si devono chiudere, si passa a chiudere al contrario, dall'ultimo ciclo inserito fino ad arrivare al primo ... Per leggere meglio le varie strutture si consiglia di scrivere le istruzioni in modo *indentato* così come si vede nell'esempio che segue.

Problema

Vogliamo calcolare il Massimo Comune Divisore (MCD) e il minimo comune multiplo (mcm) di due numeri interi a e b.

Soluzione

Possiamo utilizzare l'algoritmo di Euclide basato su divisioni successive. Se $a > b$, si ha $MCD(a,b) = MCD(b,r_1)$ dove r_1 è il resto della divisione a/b. Si cerca allora il MCD di r_1 e b applicando la stessa strategia e si va avanti in questo modo, fino a quando non si arriva ad un resto che è zero. Infatti da $a = q_1 b + r_1$, se $r_1 = 0$ allora si ha MCD(a,b)=b e siamo arrivati alla soluzione. Se $r_1 \neq 0$, sarà una quantità $0 < r_1 < b$. Ora $MCD(a,b) = MCD(b,r_1)$

$d = MCD(a,b)$ divide sia a, sia b, quindi divide sicuramente anche r_1 (che è combinazione lineare di a e b). $f = MCD(b,r_1)$ divide sia r_1 che b e quindi dividerà anche a (che è combinazione lineare di b e r_1). Ma allora $d = f$!

Ripetendo il procedimento appena descritto tra r_1 e b avremo $b = q_2 r_1 + r_2$ con $0 \leq r_2 < r_1$. Se $r_2 = 0$ siamo arrivati alla soluzione (cioé $MCD(a,b) = MCD(b,r_1) = r_1$ altrimenti sappiamo che $MCD(a,b) = MCD(r_2,r_1)$. Si andrà avanti in questo modo, ottenendo resti via via più piccoli nelle nostre divisioni, e ad un certo punto arriveremo ad un resto uguale a zero....

Usiamo questo algoritmo per calcolare $MCD(a,b)$. Per il minimo comune multiplo, invece, useremo la formula per cui

$$mcm(a,b) = \frac{ab}{MCD(a,b)}$$

```
% script per calcolare Massimo Comune Divisore (MCD) e
% minimo comune multiplo (mcm) di due numeri a e b
% Si utilizza l'algoritmo di Euclide per calcolare MCD
% e si applica la formula mcm=(ab)/MCD per il mcm
%
% funzioni che utilizziamo presenti in MATLAB/Octave sono
% -- max e min (per calcolare il massimo e il minimo tra a e b
% -- la funzione rem che fornisce il resto della divisione tra i
% due numeri dati in input
%     rem(x,y) e' il resto della divisione x/y
disp('scrivere ora due numeri a e b di cui calcolare MCD e mcm')
a=input('scrivi il valore a ');
b=input('scrivi il valore b ');
M=max(a,b);  % M = massimo tra a  b
```

```
m=min(a,b);   % m = minimo tra a e b
if rem(M,m)==0    % se il resto della divisione M/m = 0
  MCD=m;
  else              % altrimenti
   r=rem(M,m);
   while r~=0       % ciclo while all'interno del ciclo if
      M=m; m=r;     % sposto il valore di m in M e il valore di r in m
       r=rem(M,m);   % calcolo il resto della divisione M/m
   end               % fine ciclo while
   MCD=m;
end      % fine ciclo if
mcm=(a*b)/MCD;  % calcolo di mcm
disp('MCD = '); disp(MCD);
disp('mcm = '); disp(mcm);
```

Notiamo come non abbiamo generato delle variabili r_1, r_2, r_3 .... ma abbiamo lavorato sempre con M, m e r ad ogni passaggio! Su questo punto torneremo più avanti e con tutti i dettagli! Per ora l'esempio ci serviva soprattutto per vedere come usare un ciclo dentro un ciclo 😶.

5.5.2 INTERRUZIONE BRUTALE

A volte può capitare di voler interrompere brutalmente l'esecuzione di una parte o di tutto un programma (nello script sulla conversione della temperatura, abbiamo usato una variabile errore e poi abbiamo visualizzato un messaggio di errore e siamo stati gentili con il nostro script). Ma se vogliamo interrompere l'esecuzione di un ciclo (for o while) o di uno script possiamo usare l'istruzione **break**.

In genere si pone questa istruzione all'interno di un ciclo if: se determinate condizioni non sono verificate e non ha senso andare avanti con le istruzioni successive, **break** fa andare alle istruzioni che si trovano dopo la **end** che chiude il ciclo incriminato 🔫. Se invece l'istruzione viene messa al di fuori di un ciclo, allora si interrompe tutto il programma.

Ricordiamo anche l'istruzione **error** che può essere usata per visualizzare un errore, mostrando sulla Command Window un messaggio di errore.

Esempi sull'uso di **break** e **error** li vedremo più avanti, all'interno di alcuni programmi.

Esiste anche il comando **continue** da usare all'interno di un ciclo for o while, che fa saltare le istruzioni di quel livello del ciclo per andare al passaggio successivo. Non useremo mai **continue** perciò si rimandano i più curiosi ad approfondire l'argomento con esempi che si trovano in rete.

5.6 FUNCTION

E arriviamo finalmente a vedere le function 👽.

In realtà le abbiamo già viste e usate, ma abbiamo viste e usate le function predefinite che troviamo in ambiente Octave: in particolare abbiamo le cosiddette *built-in* function (di cui non possiamo vedere il file che le genera) e le function di cui

possiamo vedere il file che le genera (file con estensione .m): esempi di built-in function sono le funzioni **sin** o **exp**; della seconda famiglia fanno parte, ad esempio, **acot** (la funzione per l'arcotangente), sind (che calcola il seno in gradi)....

Riprendiamo il nostro primo script, `script0.m`, ricordiamoci che vogliamo calcolare le medie dei voti del primo anno di corso e che vogliamo anche vedere quanti libri leggono in media all'anno i nostri amici/conoscenti.

Ci serve creare *qualcosa* che calcoli la media aritmetica di un certo numero di dati e che possa essere usato quante volte vogliamo. Ci serve, appunto, creare una function!

- Una function ci permette di risolvere un algoritmo dando in input i dati che servono all'algoritmo e ottenendo in ouput i risultati.

- Una function può essere richiamata all'interno di uno script inserendo i dati di input che servono e salvando i risultati nelle variabili di output che servono allo script.

- Una function può essere richiamata da più script (oppure nella Command Window) e facilita la soluzione di tanti problemi.

Per scrivere una function, usando l'editor di testo, dobbiamo ricordarci di usare una particolare sintassi all'inizio e alla fine. Una function inizia con le righe

function [variabili di output]=nomefunction(variabili di **input**)

Dobbiamo perciò dire che stiamo scrivendo una function (scriviamo subito **function**); Tra parentesi quadre scriviamo il nome delle variabili di uscita della function, poi il nome che diamo alla function (nomefunction) e infine, tra parentesi tonde, scriviamo la lista delle variabili di input.

Dopo questa riga, conviene mettere dei commenti per dire cosa fa la function: questo è utile perchè, dalla Command Window, noi potremo scrivere **help** nomefunction e vedremo tutte le righe di commento scritte prima della prima istruzione della function! !!

Il corpo della function è fatto da istruzioni che servono per risolvere un determinato problema (mediante istruzioni sequenziali o cicli).

Quando la function finisce, si scrive **end** come ultima riga.

Il file scritto va salvato con il nome `nomefunction.m`.

ATTENZIONE Non possiamo dare nomi diversi al file e alla function. Se la function si chiama fishcake, il file in cui abbiamo scritto la function deve chiamarsi `fishcake.m`.

5.6.1 Primo esempio (semplice)

Scriviamo quindi una function che calcoli la media aritmetica di un certo numero n di dati. I dati possono essere i voti dei tre esami del primo semestre, o i voti di quindici esami, o il numero di libri letti dai nostri trenta amici... Ci conviene, quindi, pensare ad un vettore di dimensione 3, 15, 30, ovvero di dimensione n.

Facciamo una prima function (che poi miglioreremo) in cui diamo in input il vettore `x` che contiene i valori di cui calcolare la media e il numero `n` che ci dice quanti sono questi valori.

La formula da applicare è

$$Media = \frac{x_1 + x_2 + x_3 + \cdots + x_n}{n} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Per fare la somma degli x_i useremo una function predefinita chiamata **sum**.

La function è la seguente

```
function media=mediaritm0(x,n)
% prima function sulla media aritmetica
% dati di input
% x= vettore di n componenti, delle cui componenti vogliamo calcolare la media
% n= dimensione del vettore
% dati di outuput
% media  = media aritmetica
media=sum(x)/n;
end
```

Questa function l'abbiamo chiamata `mediaritm0` e quindi il file che abbiamo scritto lo abbiamo salvato con il nome `mediaritm0.m`

Possiamo usare questa function all'interno di uno script o direttamente dalla Command Window.

Intanto vediamo che scrivendo `help mediaritm0` ritroviamo sulla Command Window i commenti che abbiamo scritto prima della prima istruzione della function

```
» help mediaritm0
'mediaritm0' is a function from the file /work/didattica/lezioni_calcolo/tutorial_octave

 prima function sulla media aritmetica
 dati di input
 x= vettore di n componenti, delle cui componenti vogliamo calcolare la media
 n= dimensione del vettore
 dati di outuput
 media  = media aritmetica
```

Usiamo questa function dalla Command Window (oppure scrivendo uno script con le istruzioni che ora diamo):

```
» x=[24 28 27]; n=3;
» media=mediaritm0(x,n)
media =  26.333
```

Oppure

```
» x=[24 28 27];
» media=mediaritm0(x,3)
media =  26.333
```

La prima volta abbiamo introdotto la variabile `n` per usare la function, la seconda volta abbiamo scritto direttamente il valore 3 tra i dati di input. Possiamo anche dare in input un vettore che chiamiamo in modo diverso rispetto a `x`. Per esempio

```
» voti=[24 28 27 30 30]; n=5;
» mediavoti=mediaritm0(voti,n)
mediavoti =  27.800
```

In questo esempio abbiamo introdotto un vettore dal nome `voti` dove abbiamo messo il risultato riportato in cinque esami. Come dati di input alla function abbiamo dato `voti` e `n` e la variabile di output l'abbiamo chiamata `mediavoti`.

🚦🆘

ATTENZIONE: se nella function abbiamo usato x come variabile di ingresso e y come variabile di uscita, NON dobbiamo usare necessariamente queste variabili per poterla usare. L'importante non è il nome che diamo alle variabili, 😒l'importante è l'ordine che diamo alle variabili (sia in input che in output). Se scambiamo, ad esempio le variabili `voti` e `n` non abbiamo il risultato che vogliamo, anzi abbiamo un messaggio di errore.

```
» mediavoti=mediaritm0(n,voti)
error: mediaritm0: operator /: nonconformant arguments (op1 is 1x1, op2 is 1x5)
error: called from
    mediaritm0 at line 8 column 6
```

Questo perchè la function si aspetta un vettore come prima variabile e uno scalare come seconda variabile. Scambiando l'ordine delle variabili, non riesce a fare ciò che deve fare.

Questa function che abbiamo scritto, tuttavia, ha un suo punto debole: se abbiamo un vettore con 100 componenti, per sapere che $n = 100$ cosa facciamo? Contiamo tutte le componenti del vettore? Diventa molto difficile 👽

Usiamo una function predefinita che ci dice la lunghezza del vettore, vale a dire quante sono le componenti del vettore. Questa function si chiama **length**. A questo punto, possiamo dare solo il vettore `x` in input alla function, che chiamiamo `mediaritm.m`.

!!Occorre prestare molta attenzione all'ordine delle variabili e al significato che hanno all'interno della function. Se avessimo due vettori come dati di input, il nome da dare ai due vettori può essere diverso rispetto a quello dato quando si scrive la function ma il loro significato deve seguire l'ordine dato alla function per evitare di ottenere dei risultati privi, però, di significato!

```
function media=mediaritm(x)
% prima function sulla media aritmetica
% dati di input
% x= vettore di n componenti, delle cui componenti vogliamo calcolare la media
% dati di outuput
% media  = media aritmetica
n=length(x);
media=sum(x)/n;
```

end

Se vogliamo calcolare la media dei libri letti in media dai nostri amici/conoscenti, diventa facile farlo. Se ci ricordiamo che Giacomo ne ha letti 3, Mario 5, Giovanni 10, Lucia 7, Sofia 8, etc etc etc... introduciamo il vettore `libriletti`

```
» libriletti=[3 5 10 7 8 6 2 9 4 4 8 10 15 3 6 9 11 1 4 5];
» medialibri=mediaritm(libriletti)
medialibri =  6.5000
```

Non abbiamo perso tempo a calcolare la lunghezza del vettore!

5.7 SECONDA RISPOSTA ALLA DOMANDA: PERCHÉ SCRIPT E FUNCTION?

Dal primo esempio che abbiamo fatto sulle function possiamo già rispondere al quesito sul perchè script e function.

Una function è molto più versatile di uno script perchè può essere usata in modo immediato ed essere richiamata in qualsiasi script o nella Command Window, lavorando nella stessa directory in cui è stata scritta la function.

Occorre fare, a questo punto, un'importante osservazione. E la facciamo partendo da ciò che possiamo verificare dall'esecuzione di uno script e di una function.

Riprendiamo lo `script0.m`. Dalla Command Window, cancelliamo prima tutte le variabili con **clear**, eseguiamo lo script e controlliamo le variabili che abbiamo:

```
» clear
» script0
La media dei voti del primo semestre e'
 26.333
» who
Variables in the current scope:

AnalisiUno  Chimica     Economia    MediaVoti
```

Tutte le variabili generate all'interno dello script sono presenti nella Command Window. Ora cancelliamo tutte le variabili e dalla Command Window eseguiamo la function `mediaritm` per calcolare la media delle componenti di `x=[2 5 8]`.

```
» clear
» x=[2 5 8];
» media=mediaritm(x)
media =  5
» who
Variables in the current scope:

media  x
```

IMPORTANTE: se scriviamo una function in una determinata directory, questa function può essere richiamata da script che si trovano nella stessa directory oppure dalla Command Window aperta in quella directory. Se vogliamo usare una function OVUNQUE, cioè in qualsiasi directory stiamo lavorando, occorre sistemare il `path`, cioè il percorso che Octave può seguire (tutte le directories in cui ci sono functions) per cercare le function da utilizzare. Ci sono delle function quali `addpath` e `rmpath` che permettono di aggiungere o togliere directories dall'elenco del `path`. Non approfondiamo l'argomento perchè è secondario rispetto allo scopo principale di questo tutorial che è quello di imparare a programmare. 😳

Ma in `mediaritm` non avevamo calcolato anche la lunghezza del vettore, nella variabile `n`? Dove è finita la variabile `n`?

Non c'è! Le variabili che sono usate all'interno di una function sono variabili locali, servono solo all'interno della function e, se non sono passate in output tra le variabili di output, vengono poi perse.

Le variabili di uno script, invece, sono variabili di tipo globale, si trovano tutte sulla Command Window: c'é il rischio che vengano usate da altri script con valori non corretti. Perciò è bene, quando si scrive uno script, cancellare tutte le variabili esistenti, in modo da evitare errori non voluti.

ATTENZIONE !!Se si sta scrivendo uno script e nella Command Window esiste una variabile chiamata, per esempio,`x` e nello script vogliamo usare una variabile `x`, può andare tutto bene se `x` viene correttamente definita. Capita, tuttavia, che mentre si sta scrivendo per la prima volta lo script, questa variabile non sia definita in modo corretto oppure ci si dimentica di definirla... però nella Command Window esiste già una `x`. Nell'eseguire lo script, viene preso allora il valore di `x` già esistente: ciò può portare a risultati giusti o sbagliati che siano, ma lo script non è corretto!!! Una volta chiusa la sessione di Octave si perde quel valore di `x` e quando si vuole rieseguire quello script, questa volta avremo di sicuro messaggi di errore oppure risultati errati (a meno che non ci accorgiamo che dobbiamo definire correttamente la variabile che non era stata definita correttamente la volta precedente!). Perciò il primo comando da scrivere quando si scrive uno script è **clear**

5.7.1 UN ALTRO ESEMPIO SULLE FUNCTION

Riprendiamo l'algoritmo per calcolare MCD e mcm di alcuni numeri. Suppponiamo che dobbiamo risolvere un problema in cui ci serve calcolare MCD e mcm di diverse coppie di numeri. Nello stesso script (non andiamo certo sulla Command Window a scrivere le istruzioni per calcolare MCD e mcm...) andremo a ripetere quelle 13 righe di istruzioni tutte le volte che ci serviranno ❓😔

E ancora, se si ripresenta il problema, dobbiamo ogni volta andarci a ricopiare le istruzioni per calcolare MCD e mcm?

Certo, alla fine impareremmo bene l'algoritmo per MCD e mcm, ma non penso che nessuno ne abbia voglia 😫

Scriviamo quindi una function che calcoli MCD e mcm di due numeri interi: in ingresso avremo due variabili (i due numeri interi) e in uscita due variabili (che ci danno MCD e mcm dei due numeri).

```
function [MCD, mcm]= MCDmcm(a,b)
% function per calcolare Massimo Comune Divisore (MCD) e
% minimo comune multiplo (mcm) di due numeri a e b
% Si utilizza l'algoritmo di Euclide per calcolare MCD
% e si applica la formula mcm=(ab)/MCD per il mcm
%
%  [MCD, mcm]= MCDmcm(a,b)
%  Dati di ingresso: a e b
%  Dati in uscita: MCD e mcm
M=max(a,b);  % M = massimo tra a  b
m=min(a,b);  % m = minimo tra a e b
 if rem(M,m)==0    % se il resto della divisione M/m = 0
   MCD=m;
    else              % altrimenti
     r=rem(M,m);
     while r~=0       % ciclo while all'interno del ciclo if
         M=m; m=r;     % sposto il valore di m in M e il valore di r in m
          r=rem(M,m);   % calcolo il resto della divisione M/m
     end                % fine ciclo while
     MCD=m;
end      % fine ciclo if
mcm=(a*b)/MCD;  % calcolo di mcm
```

end

Osserviamo bene che il corpo della function non cambia rispetto allo script che avevamo fatto per calcolare MCD e mcm di due numeri. I dati di input, come vediamo non vengono passati all'interno della function ma tra i dati di input!

Scriviamo ora uno script che considera prima la coppia di dati $(15,40)$, poi $(320,456)$ e calcola di queste coppie MCD e mcm.

```
% script per calcolare MCD e mcm di diverse coppie
% di numeri.
clear
a=15; b=40;
[MCM1, mcm1]=MCDmcm(a,b);
c=320; d=446;
[MCM2, mcm2]=MCDmcm(c,d);
disp('I risultati sono')
disp('coppia di valori        MCD, mcm')
disp([a, b, MCM1, mcm1])
disp([c, d, MCM2, mcm2])
```

Se eseguiamo (lo script lo abbiamo chiamato `scriptsuMCDmcm1.m` abbiamo

```
» scriptsuMCDmcm1
I risultati sono
coppia di valori        MCD, mcm
    15    40     5   120
     320     446       2   71360
»
```

La visualizzazione dei risultati non è tanto carina. Vedremo con calma come fare di meglio. Per ora usiamo la function **disp** che è molto semplice da utilizzare.

Osserviamo che, nello script, chiamiamo due volte la stessa function (la `MCDmcm` che abbiamo creato prima), con diversi dati di input e dando nomi diversi alle variabili.

Scriviamo ora un altro script che calcola MCD e mcm delle seguenti coppie di numeri: $(60,27)$, $(75,15)$, $(84,98)$, $(144,54)$.

```
% script per calcolare MCD e mcm di diverse coppie
% di numeri.
clear
DATI=[60 27; 75 15; 84 98; 144 54];
% le coppie di dati sono 4
ncoppie=4; % per il momento scriviamo noi
% che le coppie sono quattro, vedremo poi che
% potremo usare una function predefinita per
% calcolare questo valore
for i=1:ncoppie
    [M(i),m(i)]=MCDmcm(DATI(i,1),DATI(i,2));
    % calcolo MCD e mcm delle coppie che trovo
    % su ogni riga della matrice DATI
    % percio' scrivo DATI(i,1), DATI(i,2)
    % il risultato lo pongo nelle componenti
```

SOS Un errore comune quando si scrive di proprio pugno una function e poi la si deve usare più volte all'interno di uno script è di scrivere più volte la stessa function ma con nomi diversi alle variabili di input e output, per il semplice fatto che la function deve avere variabili di input e output che cambiano!!!! 😈😱

SOS Un altro errore comune 😈😱 è di scrivere all'interno della function i valori da assegnare alle variabili di input !!Ma operando in questo modo la function darà risultati buoni solo per quei valori scritti all'interno della function!!! Non si può scrivere **function** y= miafunction(var1, var2) e poi all'interno della function scrivere, per esempio, var1=10; var2=−2 Le variabili date in ingresso vengono sovrascritte. Non si può fare!!!

```
    % i-sime di due vettori M e m (che corrispondono
    % a MCD e mcm delle coppie di dati)
end
disp('I risultati sono')
disp('coppia di valori         MCD, mcm')
for i=1:ncoppie
    disp([DATI(i,1) DATI(i,2), M(i), m(i)])
end
```

Osserviamo alcuni punti importanti di questo script (importanti per evitare errori 🚦🆘):

- per memorizzare i dati abbiamo creato una matrice di due colonne in modo da avere in ogni riga la coppia dei dati di cui calcolare MCD e mcm;
- all'interno del ciclo for la function MCDmcm viene richiamata un certo numero ncoppie (per noi uguale a 4) di volte. I dati di input e di output della function cambiano ogni volta perchè dipendono dall'indice `i` che va a prendere la coppia di dati da inserire dalla matrice e pone i risultati in due vettori (diversi perchè maiuscolo e minuscolo contano per due: M,m);
- la function MCDmcm non è stata toccata. Una volta scritta è quella: non va adattata al problema da risolvere.
- Una volta definita la variabile ncoppie abbiamo lavorato con questa e non con il valore da esso assunto (cioè 4): è molto importante lavorare il più possibile con variabili perchè se vogliamo cambiare i dati di ingresso del problema, ci sono da cambiare poche istruzioni legate ai dati di ingresso (nel nostro caso possiamo cambiare la matrice DATI e ncoppie ma una volta cambiati questi dati, non dobbiamo fare altre modifiche; se invece avessimo scritto 4 ovunque al posto di ncoppie dovremmo poi correggere molte istruzioni del nostro script per aggiornare il cambiamento legato al cambiamento dei dati....)

```
» scriptsuMCDmcm2
I risultati sono
coppia di valori         MCD, mcm
    60    27     3   540
   75   15   15   75
    84    98    14   588
   144    54    18   432
```

6

Matrici e vettori (parte prima)

3. Il successo ti procurerà nemici veri e falsi amici. ABBI SUCCESSO COMUNQUE.
4. Il bene che fai oggi verrà dimenticato domani. FAI COMUNQUE DEL BENE.
Kent M. Keith

Per poter procedere in modo più spedito nella strada della programmazione, dobbiamo spendere qualche parola su matrici e vettori in modo da poterli usare non tanto per risolvere problemi di algebra lineare quanto come strutture per memorizzare variabili che hanno lo stesso significato (ad esempio useremo il vettore degli scarti le cui componenti daranno lo scarto o la norma del vettore scarto ad ogni iterazione di un metodo iterativo). Conviene quindi sapere alcune informazioni utili.

6.1 Funzioni vettorizzate

Quando lavoriamo con le funzioni matematiche predefinite, queste sono vettorizzate.

Cosa significa *vettorizzate* ??

Vuol dire che la funzione matematica può essere valutata in un vettore o in una matrice e dare come risultato il vettore o la matrice con i valori della funzione nei corrispondenti valori del vettore o matrice di partenza.

Vediamo un esempio.

Se dobbiamo valutare e^2, e^{10}, e^{-2}, e^4, la prima cosa che ci verrebbe in mente di fare potrebbe essere la seguente (nell'ipotesi che i risultati ci servano tutti e non dobbiamo sovrascriverli):

```
» x1=2; y1=exp(x1);
» x2=10; y2=exp(x2);
» x3=-2; y3=exp(x3);
» x4=4; y4=exp(x4);
» disp([y1 y2 y3 y4])
   7.3891e+00   2.2026e+04   1.3534e-01   5.4598e+01
```

Abbiamo visualizzato i risultati tramite la function **disp**. La stessa cosa, tuttavia, può essere fatta in questo modo: creiamo

un vettore che ha componenti $[2, 10, -2, 4]$ e poi valutiamo la funzione esponenziale nel vettore, componente per componente.

```
» x=[x1 x2 x3 x4]; y=exp(x);
» disp(y)
   7.3891e+00   2.2026e+04   1.3534e-01   5.4598e+01
```

Il risultato è lo stesso ma è stato ottenuto in modo molto più elegante e compatto 😯.

Stesso discorso vale se lavoriamo con matrici:

```
» A=[x1 x2; x3 x4]; B=exp(A);
» disp(B)
   7.3891e+00   2.2026e+04
   1.3534e-01   5.4598e+01
```

6.1.1 Operazioni elementari vettorizzate

È molto importante ora capire bene quanto diremo 😔

Se vogliamo fare operazioni di addizione o sottrazione tra vettori o tra matrici pensando di fare le operazioni componente per componente (che è quello che si fa usualmente anche in algebra lineare), possiamo usare tranquillamente gli operatori + e −.

```
» x=[1 2 3 4]; y=[5 6 7 8];
» a=x+y
a =

    6    8   10   12

» b=x-y
b =

   -4   -4   -4   -4
» A=[1 2; 3 4]; B=[5 6; 7 8];
» C=A+B
C =

    6    8
   10   12

» D=A-B
D =

   -4   -4
   -4   -4
```

Ma se dobbiamo fare operazioni di moltiplicazione, divisione ed elevamento a potenza componente per componente (⚠) non

possiamo più usare gli operatori `*`, `^` perchè questi vengono usati così come sono per operazioni di algebra lineare (che vedremo più avanti). Se vogliamo fare moltiplicazione e divisione tra matrici (e i vettori sono casi particolare di matrici) tra elementi corrispondenti, dobbiamo usare gli operatori `.*` e `./`. Stessa cosa se vogliamo fare elevamento a potenza di una matrice elevata ad un'altra matrice, tra elementi corrispondenti: usiamo l'operatore `.^`.

Vediamo degli esempi 😮Vogliamo fare questi prodotti: $1*5$, $2*6$, $3*7$, $4*8$: se abbiamo i vettori $[1,2,3,4]$ e $[5,6,7,8]$, l'operazione di moltiplicazione componente per componente ci darà precisamente quello che vogliamo.

```
» x=[1 2 3 4]; y=[5 6 7 8];
» x.*y
ans =

    5   12   21   32
```

Allo stesso modo se vogliamo fare divisioni componente per componente scriviamo

```
» x./y
ans =

   0.20000   0.33333   0.42857   0.50000
```

Per l'elevamento a potenza, componente per componente:

```
» x.^y
ans =

       1      64    2187   65536
```

Stesso discorso vale per matrici. Se prendiamo le matrici `A` e `B` usate in precedenza, abbiamo

```
» A.*B
ans =

    5   12
   21   32

» A./B
ans =

   0.20000   0.33333
   0.42857   0.50000
» A.^B
ans =

       1      64
    2187   65536
```

6.2 PRE-ALLOCAZIONE DI MEMORIA

Negli script e function che scriveremo, non abbiamo bisogno di dichiarare tutte le variabili come accade in altri linguaggi di programmazione (una cosa in meno da imparare 😀). A volte però risulta utile inizializzare delle variabili la cui dimensione a priori non è nota.

Vediamo un esempio (che ci servirà in seguito): se dobbiamo implementare un metodo iterativo, a priori non sappiamo se faremo 5, 30 o il valore massimo di iterazioni che fisseremo problema per problema (questa variabile la chiameremo `itmax`). Per ogni iterazione effettuata, andremo a conservare il valore dello scarto (o della norma dello scarto) in un vettore, che avrà dunque una lunghezza che dipende dal numero di iterazioni eseguite.

Nulla ci vieta di partire da un vettore di lunghezza unitaria e poi di aggiungere via via le componenti ad ogni iterazione. Non facciamo nessun errore, però ⚠‼questo può comportare un rallentamento nell'esecuzione del programma.

Certamente per i programmi che faremo noi questo rallentamento non sarà così evidente in quanto risolveremo problemi semplici, ma se le dimensioni in gioco sono grandi, le differenze di tempi di esecuzione saranno fondamentali.

Perchè succede questo? Perchè quando viene eseguito un programma (script o function) deve essere riservato lo spazio in memoria per le diverse variabili in gioco. Se una variabile è scalare lo spazio in memoria è quello di una variabile scalare, se c'è una matrice di dimensione $n \times m$ lo spazio in memoria è riservato per una matrice $n \times m$. Per semplicità di comprensione, supponiamo che per ciascuna variabile scalare lo spazio di memoria che occupa sia una celletta, e se la variabile è matriciale, lo spazio di memoria occupato sia di $n \times m$ cellette, tutte vicine tra loro.

Se incominciamo a lavorare con una variabile scalare, ma poi questa variabile diventa un vettore di componenti 2, 3, 4, ... ogni volta occorrerà risistemare lo spazio in memoria occupato dal vettore perchè si aggiunge una componente alla volta! Bisognerà riordinare ogni volta le cellette dello spazio di memoria perchè quelle del vettore (la cui lunghezza aumenta sempre più) devono essere tutte vicine tra loro!

Se all'inizio avessimo detto che quella variabile sarebbe stato un vettore di al più $nmax$ componenti, già da subito sarebbe stato organizzato lo spazio di memoria con $nmax$ cellette riservate per quella variabile, comporando tanto lavoro in meno e quindi maggiore velocità di esecuzione.

Si parla perciò di preallocazione quando si inizializza una variabile ad una dimensione maggiore o uguale di quella che sarà effettivamente.

In genere noi creeremo delle matrici di tutti zeri, usando una funzione predefinita e poi taglieremo la matrice alla dimensione effettiva.

La funzione predefinita che useremo sarà **zeros**. Facciamo

zeros(m,n) crea una matrice di m righe e di n colonne con componenti tutte uguali a zero.

un esempio creando un vettore di tutti zero di 10 componenti

```
» nmax=10; vett=zeros(nmax,1)
vett =

   0
   0
   0
   0
   0
   0
   0
   0
   0
   0
```

Ora diamo un valore solo alle prime quattro componenti: deve essere $vett(i) = i$. Possiamo quindi porre il sottovettore di componenti da 1 a 4 uguale al vettore di componenti [1,2,3,4].

```
» n=4; vett(1:n)=1:n
vett =

   1
   2
   3
   4
   0
   0
   0
   0
   0
   0
```

Le componenti con $i > n$ non ci interessano più. Perciò tagliamo il vettore prendendo solo le prime n componenti:

```
» vett=vett(1:n)
vett =

   1
   2
   3
   4
```

⚠!!ATTENZIONE: si osservi il diverso uso di vett(1:n) se messo a sinistra o a destra del segno di assegnazione.

6.3 LUNGHEZZA E DIMENSIONE DI MATRICI E VETTORI

Abbiamo visto la function predefinita **length** associata alla lunghezza di un vettore.

Se questa function è applicata ad una matrice, ci restituisce la dimensione massima della matrice stessa.

```
» A=[1 2 3; 4 5 6]
A =

   1   2   3
   4   5   6

» length(A)
ans =  3
```

È utile conoscere un'altra function che restituisce le dimensioni di una matrice: la function si chiama **size**.

```
» size(A)
ans =

   2   3

» x=[1 2 3]; size(x)
ans =

   1   3

» x=x'; size(x)
ans =

   3   1
```

7

FUNZIONI MATEMATICHE E GRAFICI

5. L'onestà e la franchezza ti renderanno vulnerabile. SII ONESTO E FRANCO COMUNQUE .
6. Anche i più grandi uomini e le più grandi idee possono essere ostacolati dagli uomini più piccoli con piccole menti. PENSA COMUNQUE IN GRANDE.
Kent M. Keith

PER RISOLVERE PROBLEMI di Calcolo Numerico avremo spesso a che fare con funzioni matematiche (approssimare zeri di funzioni, calcolare integrali di funzioni, approssimare funzioni....). Nelle pagine che seguono vedremo un modo molto semplice per definire funzioni matematiche.

In genere, data una funzione siamo curiosi di vederne il grafico, perciò vedremo anche come fare grafici (non solo di funzioni).

7.1 FUNZIONI MATEMATICHE

Ci sono diverse modalità per definire funzioni matematiche. Se lavoriamo con funzioni scalari (dipendenti dalla sola variabile x) e se la funzione cambia a seconda dell'intervallo in cui si trova x, scriveremo una function per la nostra funzione. Se invece la funzione si scrive utilizzando un'unica scrittura ($f(x) = \sin(x^2+3) + e^{5x}$, $f(x) = \tan(x-2)$, etc.) allora useremo una delle possibili strategie per rappresentarla. Descriviamo la strategia che ci sembra più semplice (altre modalità di rappresentazione delle funzioni matematiche sono in via di *estinzione* in quanto stanno diventando obsolete e quindi non ne parleremo.

Ci sono già tante funzioni predefinite in Octave: pensiamo a **sin**,**cos**, **exp**, **log**, **log10**, **tan**, ... Queste funzioni possono essere valutate in variabili scalari ma anche in vettori (e matrici). Perciò possiamo scrivere in modo del tutto indifferente

```
» x=10;
» sin(x)
ans = -0.54402
» x=[1 2 3 4];
» sin(x)
ans =

   0.84147   0.90930   0.14112  -0.75680
```

Il risultato della funzione è dato componente per componente. Si dice che la funzione è vettorizzata.

Consideriamo ora altre funzioni reali che dipendono da una sola variabile x.

Ad esempio:

- $f(x) = x^2 + e^x$

- $y = \sqrt{1-x^2}$
- $g(x) = \ln(5-x)$
- ...

La funzione l'abbiamo chiamata $f(x)$ o y (si tratta sempre però di una funzione che dipende da x, $y = y(x)$) o $g(x)$).

Cosa facciamo in Octave per poter lavorare con queste funzioni? Vediamo dalla Command Window.

```
» f=@(x) x.^2 +exp(x)
f =

@(x) x .^ 2 + exp (x)

» y= @(x) sqrt(1-x.^2);
» g= @(x) log(5-x);
```

Introduciamo il nome che vogliamo dare alla funzione, per esempio `f`, questa sarà la variabile che rappresenta la funzione. Dopo il segno di uguaglianza, scriviamo il simbolo @ (at,chiocciolina) e, tra parentesi, diciamo che la variabile da cui dipende la funzione è `x` e poi scriviamo come è fatta la funzione che dipende da `x`. Facendo **whos** abbiamo

```
» whos f  y  g
Variables in the current scope:

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        f           1x1                          0  function_handle
        y           1x1                          0  function_handle
        g           1x1                          0  function_handle

Total is 3 elements using 0 bytes
```

Abbiamo creato delle *function handle*. Una volta create, le function handle possono essere usate come le function matematiche già esistenti. Così come scriviamo

```
» sin(pi/2)
ans =  1
» a=2; sin(a);
» x=10; sqrt(x);
» c=log(a+2);
```

in modo del tutto analogo possiamo fare

```
» f(4)
ans =  70.598
» a=0.5; y(a)
ans =  0.86603
» c=g(a)
c =  1.5041
```

Dobbiamo prestare attenzione a due cose importanti:

1. Se abbiamo definito la function handle nella Command Window, quando chiudiamo la sessione di Octave perdiamo la function. Se ci servirà la volta successiva, dobbiamo ridefinirla. Quindi conviene definire le function handle in script in modo da poterle richiamare facilmente tramite lo script.

2. Se si pone attenzione agli esempi fatti, le function handle sono state scritte in modo da avere le operazioni di moltiplicazione, divisione ed elevamento a potenza in forma vettorizzata (abbiamo scritto `x.^2`: questo è utile per poter valutare la function handle in vettori e per poter fare facilmente il grafico della funzione).

Non è necessario che sia chiamata `x` la variabile di input della function handle.

⚠ATTENZIONE: Le operazioni di moltiplicazione, divisione ed elevamento a potenza vengono vettorizzate.

7.2 Grafici di funzione

Se dobbiamo fare il grafico di una funzione, scritta come function handle, possiamo sfruttare la function predefinita chiamata ezplot.

Vediamo subito un esempio

```
» y= @(x) x.*log(4-x.^2);
» ezplot(y)
```

Con questi comandi viene creata la figura 7.1

La function ezplot, di default, fa il grafico della funzione che viene passata tra i suoi dati di input, per x che varia nell'intervallo $]-2\pi, 2\pi[$. Sempre di default la figura ha come titolo il nome della funzione di cui viene fatto il grafico. Se vogliamo fare il grafico della funzione in un preciso intervallo, dobbiamo precisare gli estremi dell'intervallo tra i dati di input della function ezplot. Vediamo come: prendiamo sempre la funzione di prima e di questa vogliamo fare il grafico per $x \in [-2, 2]$.

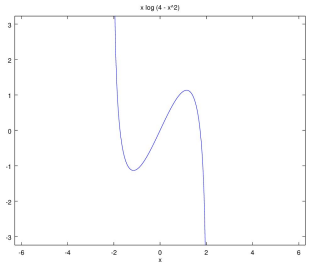


Figura 7.1: Uso di ezplot

Se invece la funzione è definita per $x \geq 0$, di default il grafico viene fatto per $x \in]0, 2\pi[$.

```
» a=-2; b=2; ezplot(y,[a,b])
```

Viene generato il grafico che vediamo in Figura 7.2.

‼Attenzione: osserviamo che il grafico precedente non c'è più. Lo abbiamo sovrascritto con questo nuovo grafico!

⚠In modo del tutto equivalente potremmo anche scrivere `» ezplot(y,[-2,2])`
Questa strada, però, conviene usarla solo se si deve fare un unico grafico. Se vanno fatti più grafici nello stesso intervallo, è meglio usare le variabili `a` e `b` (o altre variabili) per definire gli estremi dell'intervallo.

7.2.1 Altri fatti importanti

Abbiamo capito come si fa il grafico di una funzione che abbiamo definito come function handle tramite il simbolo @ della chiocciolina.

Vediamo ora altri dettagli.

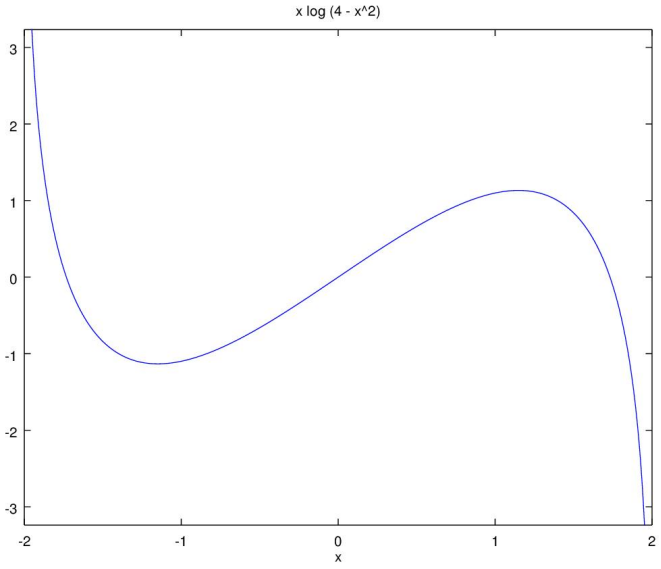


Figura 7.2: `» a=-2; b=2; ezplot(y,[a,b])`

- Se vogliamo fare il grafico di una funzione predefinita (ad esempio `sin,cos, exp, ...` non possiamo scrivere `ezplot(sin)` ma `ezplot(@sin)`: rendiamo la funzione predefinita una funzione di tipo handle. Oppure dobbiamo scrivere `ezplot('sin')` (scrivere il nome tra apici).

- Se abbiamo una funzione matematica scritta come function e vogliamo fare il grafico di questa funzione, ci rifacciamo al punto precedente. Vediamo con un esempio

Scriviamo la function dal nome `funesercizio.m`

```
function y=funesercizio(x)
% funzione matematica scritta come function
% dati di input x (scalare o vettore)
% dati di output y=f(x)
% dove f(x)= sqrt(4-x^2); se x in [-2,2]
%       f(x)= x+2    se x <= -2
%       f(x)= x-2    se x>= 2
% x puo' essere un vettore
n=length(x);
for i=1:n
if x(i)<=2 && x(i)>=-2
   y(i)= sqrt(4-x(i)^2);
elseif x(i) <= -2
   y(i)=x(i)+2;
else
   y(i)=x(i)-2;
end
end
end
```

Osserviamo che questa funzione non potremmo scriverla come function handle in quanto dipende da un ciclo if. Scriviamo la

funzione in modo da poterla valutare su vettori (perciò controlliamo la lunghezza della variabile `x` e andiamo a controllare ogni componente di `x` per assegnare il giusto valore alla funzione). Per fare il grafico della funzione possiamo usare la funzione ezplot. In modo equivalente, possiamo usare i due comandi:

```
» ezplot(@funesercizio)
» ezplot('funesercizio')
```

Si genera il grafico di Figura 7.3.

La function ezplot si può usare solo se la funzione è vettorizzata, altrimenti abbiamo un messaggio di errore. 😒

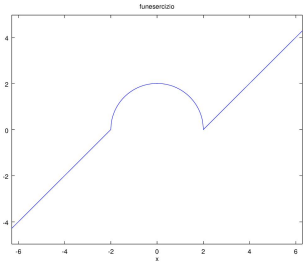


Figura 7.3: Uso di ezplot con una funzione data da function.

Facciamo un esempio prendendo un'altra funzione, ma scrivendola in forma non vettorizzata. Sia la funzione data da $f(x) = \frac{1}{1+25x^2}$, Intanto guardiamo bene la funzione di cui fare il grafico. Abbiamo una certa quantità al denominatore: dobbiamo usare le parentesi ‼Scriviamo la function handle, usiamo ezplot e vediamo che non si crea nessun grafico, anzi abbiamo un messaggio di errore. La causa è dovuta al fatto che la function non è vettorizzata!

```
» f=@(x) 1/(1+25*x^2);
» ezplot(f)
error: for A^b, A must be a square matrix. Use .^ for elementwise power.
error: called from
     at line -1 column -1
    __ezplot__ at line 382 column 13
    ezplot at line 76 column 19
```

Vettorizziamo la function

```
» f=@(x) 1./(1+25*x.^2);
» ezplot(f)
```

Ora viene creato il grafico (si veda Figura 7.4).

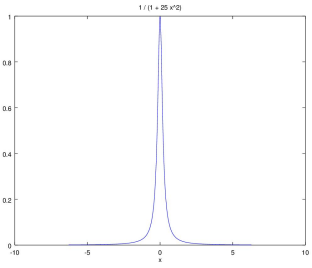


Figura 7.4: Grafico di $f(x) = \frac{1}{1+25x^2}$

Se è complicato scrivere la funzione in forma vettorizzata (😀quindi per funzioni non handle), o se vogliamo fare altri grafici ma non con funzioni, che cosa possiamo fare per risolvere il problema? Possiamo fare un grafico per coppie di punti, usando la function **plot**.

7.3 GRAFICI CON PLOT

Se abbiamo delle coppie di punti (x_i, y_i) $i = 1, 2, \ldots, n$ e vogliamo disporle su un grafico, dobbiamo avere due vettori: il vettore che contiene tutte le ascisse e il vettore che contiene tutte le ordinate.

Siano date ad esempio le coppie $(-1, 2)$, $(1, 0)$, $(3, 5)$, $(4, 7)$. Vogliamo disporre questi punti, come punti, sul piano cartesiano. Operiamo in questo modo

```
» x=[-1 1 3 4];
» y=[2  0 5 7];
» plot(x,y,'o')
```

Abbiamo creato il vettore `x` che contiene le ascisse dei punti assegnati, il vettore `y` che contiene le ordinate dei punti assegnati. Infine abbiamo scritto **plot**(x,y,'o') per dire di fare il grafico in cui le ascisse sono date nel vettore `x`, le ordinate in `y` e il grafico va fatto usando un cerchietto (`'o'`) per rappresentare i punti. Vediamo tutto in Figura 7.5

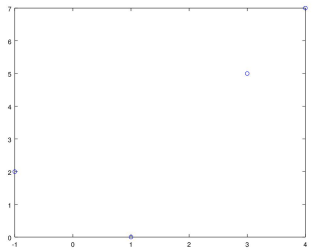


Figura 7.5: Grafico per punti con **plot**

Se togliamo la modalità dei cerchietti, possiamo scrivere semplicemente:

```
» plot(x,y)
```

In tal caso il grafico è quello in Figura 7.6 dove i punti vengono uniti tramite linee.

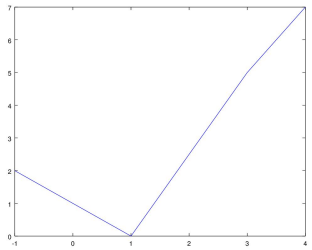


Figura 7.6: Grafico per linee con **plot**

La funzione **plot** può essere usata per fare grafici di funzione. Basta creare un vettore di ascisse prendendo punti equidistanti nell'intervallo assegnato, valutare la funzione in questi punti ricavando il vettore delle ordinate e infine applicare la function **plot**. Per creare il vettore di punti equidistanti usiamo la function predefinita che si chiama **linspace**

Vediamo un esempio con la funzione scritta prima, per $x \in [-1,1$ 😏(si veda Figura 7.7).

```
» x=linspace(-1,1); y=f(x); plot(x,y)
```

⚠‼😫Quando si usa **plot** per fare grafici di funzione, occorre prestare molta attenzione al dominio di definizione, altrimenti c'è il rischio di fare grafici nel campo complesso e non nel campo dei reali!!!! Vediamo un esempio con la funzione logaritmo naturale!

x=linspace(a,b,n) crea il vettore `x` di n componenti equidistanti tra `a` e `b`; **x=linspace**(a,b) crea automaticamente un vettore con 100 componenti.

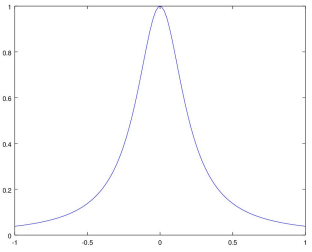


Figura 7.7: Grafico della funzione $f(x) = \dfrac{1}{1+25x^2}$ con **plot**

```
» ezplot('log')
» ezplot('log',[-2 2])
» x=linspace(-2,2); y=log(x);
» plot(x,y)
» y
y =

 Columns 1 through 3:

   0.69315 + 3.14159i   0.67274 + 3.14159i   0.65190 + 3.14159i

 Columns 4 through 6:

   0.63063 + 3.14159i   0.60889 + 3.14159i   0.58666 + 3.14159i
.....(altre righe che non riportiamo)
```

Con ezplot il grafico viene fatto nel campo dei reali e quindi là dove la funzione non è definita, non viene fatto nessun grafico.

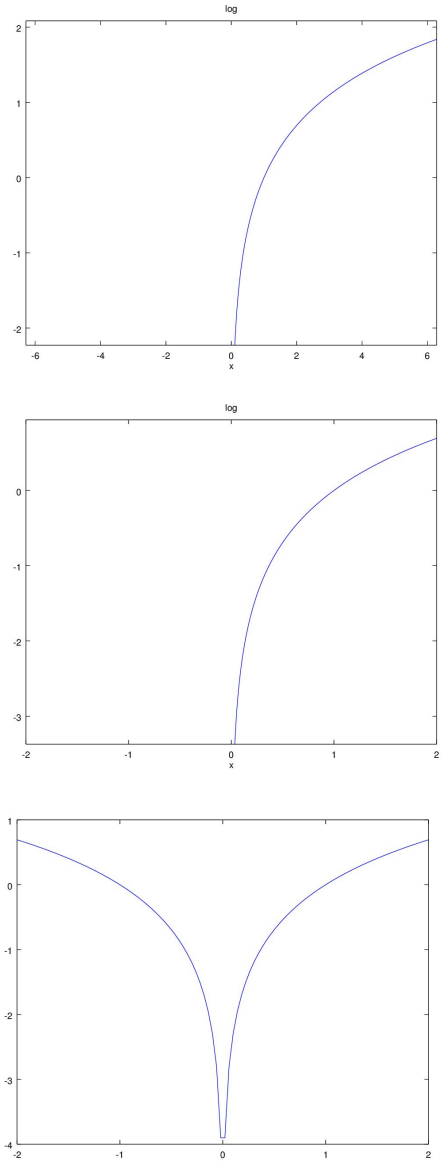


Figura 7.8: ⚠Attenzione all'insieme di definizione delle funzioni.

Ma se noi andiamo a valutare la funzione logaritmo per valori negativi, (quindi dove non abbiamo risultati reali) veniamo *trasferiti* nel campo dei numeri complessi e quindi il grafico non è quello che ci aspettiamo! Perciò prestare molta attenzione all'insieme di definizione! 🕶

7.4 Fare bei grafici

Per rendere un grafico bello 🌻🌞abbiamo a disposizione tanti elementi su cui giocare: il colore, lo spessore delle linee, mettere insieme più curve, aggiungere titoli, ...

Ora vedremo qualcosa di tutto quello che si può fare, e lo vedremo con degli esempi. Consideriamo due vettori di ascisse e di corrispondenti ordinate di cui fare il grafico tramite **plot**.

```
» a=2; b=5;
» x=linspace(a,b,12); y=x.^2+3;
```

- Vogliamo fare il grafico sia per linee che per cerchietti. Le seguenti istruzioni sono tutte equivalenti tra loro.

```
» plot(x,y,'-',x,y,'o')
» plot(x,y,x,y,'o')
» plot(x,y,'o',x,y)
```

Viene generato il grafico di Figura 7.9 (in alto). Cosa abbiamo fatto? Abbiamo dato come parametri di input i due vettori ascisse-ordinate specificando come metterli nel grafico: `'-'` linea, `'o'` cerchietto. Possiamo cambiare il modo di fare le linee, considerando linea-punto `'-.'`, linea tratteggiata `'--'`, linea a puntini `':'`. Possiamo cambiare il modo di fare la curva per puntini, mettendo, al posto dei cerchietti, altri simboli: `'+'`, `'*'`, `'.'` sono i principali. Per vedere tutte le potenzialità, scrivere `help plot` sulla Command Window. Facciamo una prova di tutto questo scrivendo

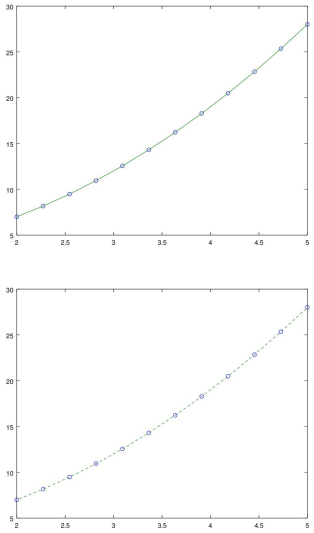


Figura 7.9: Varie modalità di uso di **plot**

```
» plot(x,y,'o',x,y,'- --')
```

Si ha il grafico di Figura 7.9 (in basso)

- Nei grafici appena fatti, notiamo che il colore cambia per ciascuna serie di vettori ascisse-ordinate. La prima serie è di colore blu, la seconda verde. Se vogliamo cambiare il colore, possiamo farlo specificando, sempre tra apici, il colore che vogliamo. In genere subito dopo la coppia dei vettori ascisse-ordinate, si mette tra apici o solo il colore o il colore e il tipo di linea o punto che vogliamo. Per i colori abbiamo queste sigle: `k`: nero, `b`: blu, `g`: verde, `y`, giallo, `m`: magenta, `r`: rosso, `w`: bianco, `c`: celeste.

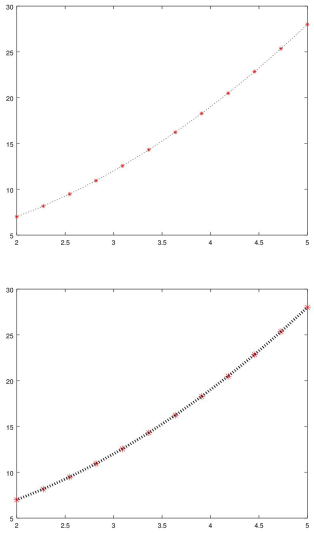


Figura 7.10: Stessi dati: risultati visivi diversi.

```
» plot(x,y,'r*',x,y,'k:')
```

Abbiamo il grafico di Figura 7.10 (in alto).

- Vogliamo cambiare lo spessore o la dimensione delle linee e dei punti? Possiamo farlo usando, le stringhe `'LineWidht'` e `'MarkerSize'` e scrivendo, dopo la virgola, il numero che corrisponde alla dimensione che vogliamo. Facciamo un esempio per capire meglio.

```
» plot(x,y,'r*','MarkerSize', 10, x,y,'k:', 'LineWidth', 4)
```

Il risultato è in Figura 7.10 (in basso).

7.4.1 CENNI SUL MODO DI MIGLIORARE EZPLOT

Volendo si può fare qualcosa anche sul colore e sullo spessore dei grafici che si hanno con ezplot. Diamo solo qualche cenno.

Quando noi usiamo ezplot (ma la stessa cosa si può dire per **plot**) noi applichiamo una funzione predefinita. Possiamo assegnare ad una variabile il risultato di questa funzione (al di là del fatto che viene generato il grafico). Possiamo quindi scrivere qualcosa del genere

```
» f=@(x) x.^3 -x.^2+x-1;
» h=ezplot(f)
h = -3.8712
» h=ezplot(f);
»
```

Assegniamo alla variabile `h` l'output della function. Se non mettiamo il punto e virgola leggiamo un numero (che non ci interessa, quindi mettiamo il punto e virgola). Ora possiamo modificare le proprietà di `ezplot`, lavorando sulla variabile `h`, usando un'altra function predefinita che si chiama **set**. Con l'esempio vediamo come usarla:

```
» set(h, 'color', 'm', 'LineWidth', 6)
```

Il risultato è visibile in Figura 7.11. In modo del tutto analogo si può lavorare anche con **plot**.

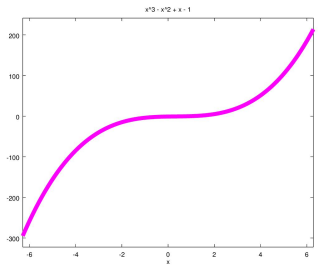


Figura 7.11: Cambiamenti di proprietà con ezplot

7.5 PIÙ GRAFICI

Supponiamo ora di avere due o più serie di ascisse-ordinate da rappresentare nello stesso grafico. Vediamo alcune delle modalità che possono essere percorse. Partiamo da due coppie di vettori ascisse-vettori (i vettori delle ascisse possono essere gli stessi o diversi)

```
» a=-1; b=4; n=30; m=20;
» x1=linspace(a,b,n); y1=x1.^2-1;
» x2=linspace(a,b,m); y2=x2-2;
```

- Primo modo: ⚠usare **plot** una sola volta e dare in input le due serie di vettori

  ```
  » plot(x1,y1,x2,y2)
  ```

- Secondo modo: Fare il primo grafico, scrivere `hold on` in modo da sovrascrivere i grafici successivi, fare il secondo grafico.

⚠ATTENZIONE: se poi non si devono fare altre grafici scrivere `hold off` per evitare di sovrascriverli in seguito.

```
» plot(x1,y1)
» hold on
» plot(x2,y2)
» hold off
```

Nel primo caso le due curve risultano di colore diverso, blu e verde, perchè abbiamo usato solo una volta **plot**. Nel secondo caso le due curve sono entrambe blu. Ma sappiamo come cambiare colore, linea, spessore...

⚠‼ATTENZIONE: prima fare il primo grafico e poi aggiungere il comando `hold on` onde evitare possibili brutti scherzi!

7.6 Grafici in scala logaritmica o semilogaritmica

Altri grafici che useremo spesso sono quelli in scala logaritmica o semilogaritmica. Per farli avremo bisogno di alcune function predefinite che si chiamano **loglog**, **semilogx**, **semilogy**: la prima effettua il grafico in scala logaritmica su entrambi gli assi (delle x e delle y), la seconda effettua il grafico in scala logaritmica sull'asse delle ascisse (x), la terza invece in scala logaritmimca sull'asse delle ordinate (y). I nomi sono facili da ricordare. L'importante è saperle usare correttamente. 😭

La prima cosa di cui accertarsi é di avere i dati in vettori di uguale lunghezza ascisse-ordinate. Non dobbiamo fare nessuna trasformazione in scala logaritmica perché la trasformazione viene fatta dalla funzione utilizzata. Quindi se `x,y` rappresentano i vettori con le ascisse e le ordinate, il comando **loglog**(x,y) farà il grafico in scala logaritmica, **semilogx**(x,y) farà il grafico in scala semilogaritmica sull'asse delle ascisse, **semilogy**(x,y) farà il grafico in scala semilogaritmica sull'asse delle ordinate.

Vediamo un esempio per tutti, prendendo spunto da algoritmi che useremo spesso in seguito. Se applichiamo un metodo iterativo e ad ogni iterazione abbiamo una misura dell'errore che stiamo commettendo, possiamo fare un grafico in scala semilogaritmica sull'asse delle ordinate per vedere il profilo di convergenza del metodo. In particolare sull'asse delle ascisse porremo le iterazioni e sull'asse delle ordinate le misure dell'errore preso ad ogni iterazione. Supponiamo, per fare un esempio, che siano state effettuate 5 iterazioni e che le misure degli errori ad ogni iterazione siano state, rispettivamente, $1.e-1$, $4.e-2$, $2.3e-2$, $1.12e-2$, $6.4e-3$. Per fare il grafico faremo qualcosa del genere:

```
» misure=[1.e-1, 4.e-2, 2.3e-2, 1.12e-2, 6.4e-3];
» iter=5;
» semilogy([1:iter], misure)
```

Abbiamo creato il vettore `[1:iter]` che ha componenti $1, 2, 3, ...iter$ in modo da poter avere le coppie ascisse-ordinate.

⚠🆘ATTENZIONE. Anche per i grafici in scala semilogaritmica o logaritmica, se si devono fare più grafici nella stessa figura, si faccia sempre il primo grafico e poi si usi `hold on` per sovrascrivere i successivi. Un errore (😱diffuso 😱) che in Octave non produce effetti disastrosi ma in MATLAB® sì, è quello di scrivere i seguenti comandi anche se non è stato fatto ancora nessun grafico:

```
hold on
semilogy([1:iter], misure)
```

In Octave la figura che si ottiene è quella corretta. In MATLAB® no ‼perchè il comando `hold on` conserva gli assi cartesiani e non si passa alla scala semilogaritmica! Quindi fare sempre il primo grafico e poi (se serve ‼) usare `hold on`.

7.7 Ulteriori dettagli

Anche per queste funzioni possiamo cambiare colore, spessore, linea, come abbiamo fatto per ezplot.

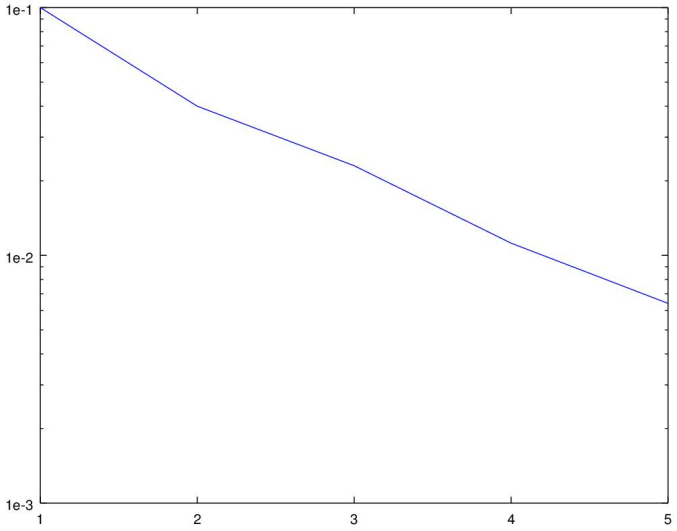


Figura 7.12: Uso di **semilogy**

Ci sarebbero tante cose da dire sui grafici. Ne riportiamo ancora alcune che possono ritornare utili 😀

7.7.1 Titolo, etichette, legenda

Possiamo aggiungere titolo, etichette sugli assi, legende. Un esempio lo facciamo dall'ultimo grafico fatto. Dopo aver creato il grafico aggiungiamo queste istruzioni:

```
» title('Esempio con semilogy')
» xlabel('iterazioni')
» ylabel('misure degli errori')
» legend('Errori')
```

Il risultato lo vediamo in Figura 7.13. Questi comandi possiamo inserirli dopo una funzione che fa un grafico (**plot**, ezplot,...). La stringa di caratteri da scrivere viene messa sempre tra apici. Se abbiamo più curve, la legenda deve riportare più stringhe che descrivono le varie curve nell'ordine in cui sono state date.

🏠. Un dettaglio. Se vogliamo inserire il contenuto di una variabile numerica nel titolo o nella legenda, dobbiamo convertirla in stringa attraverso la function **num2str**. Ad esempio `title(['Esempio con ' num2str(iter) ' iterazioni'])` darà il titolo `Esempio con 5 iterazioni`. In questo caso abbiamo creato un vettore (ci sono le parentesi quadre) di stringhe di caratteri.

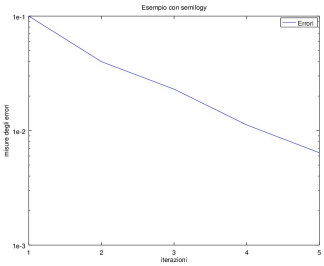


Figura 7.13: Titolo, etichette, legende.

7.7.2 Subplot

Possiamo creare una figura che ha al suo interno più grafici disposti su una griglia di m righe e n colonne. Si fa tutto questo con la function **subplot**.

Vediamo come fare con degli esempi.

- Consideriamo due funzioni che vogliamo mettere in due grafici ma nella stessa figura. Vogliamo mettere sulla stessa colonna i due grafici. Opereremo in questo modo (usiamo ezplot ma possiamo usare **plot** o qualsiasi altra funzione che fa un grafico).

```
» subplot(2,1,1)
» ezplot(@exp)
» subplot(2,1,2)
» ezplot(@log)
```

L'istruzione **subplot**(2,1,1) dice che vogliamo disporre i grafici su 2 righe e 1 colonna e il primo grafico che faremo sarà quello indicato nell'istruzione successiva. L'istruzione **subplot**(2,1,2) dice che vogliamo disporre i grafici su 2 righe e 1 colonna e il secondo grafico sarà quello indicato nell'istruzione successiva.

Il risultato lo vediamo in Figura 7.14. Quindi **subplot**(n,m,i) dice che creiamo una griglia di n righe e m colonne e l'i-simo grafico è quello che viene fatto all'istruzione successiva.

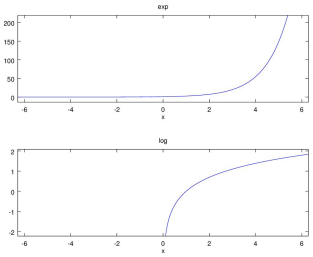


Figura 7.14: Esempio di **subplot** su una colonna

- Ora disponiamo i grafici su una riga e due colonne. Osserviamo che non è necessario seguire l'ordine crescente per mettere i grafici. Lo vediamo con questo esempio il cui risultato è in Figura 7.15.

```
» subplot(1,2,2)
» ezplot(@sin)
» subplot(1,2,1)
» ezplot(@cos)
```

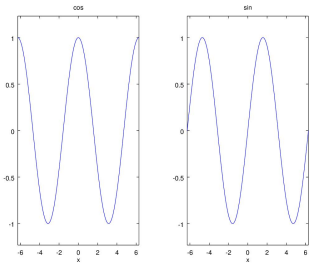


Figura 7.15: Esempio di **subplot** su una riga

- Infine disponiamo quattro grafici su due righe e due colonne.

```
» x=linspace(-2,2,10); y=sqrt(4-x.^2);
» subplot(2,2,1)
» plot(x,y,'*','MarkerSize',6);
» subplot(2,2,2)
» plot(x,y,'k', 'LineWidth', 4)
» subplot(2,2,3)
» plot(x,y,'o',x,y,'-.')
» subplot(2,2,4)
```

I grafici vengono disposti riga dopo riga andando da sinistra verso destra. Il risultato lo vediamo in Figura 7.16.

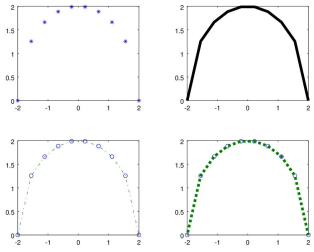


Figura 7.16: Esempio di **subplot** su due righe e due colonne

7.7.3 APPROFONDIMENTI

Se si vogliono creare più figure distinte, allora si apre una nuova figura mediante la function **figure**.

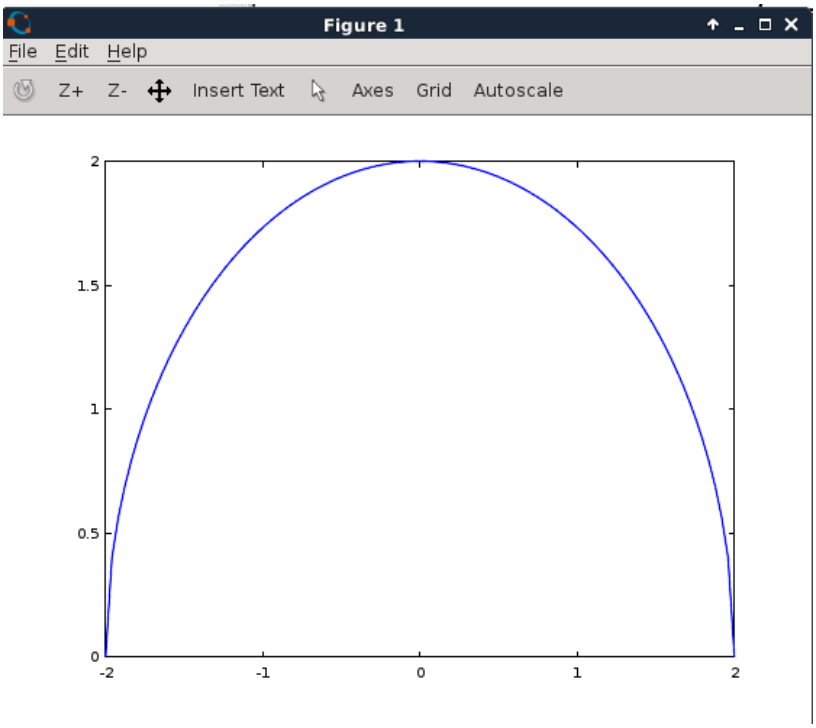


Figura 7.17: Finestra con una figura

Quando noi creiamo una figura, la finestra della figura riporta la dicitura `Figure 1` (si veda la Figura 7.17). Perciò se vogliamo creare un'altra figura possiamo scrivere

```
» figure(2)
```

e si aprirà una finestra con la figura 2 vuota (si veda la Figura 7.18). Le istruzioni di grafica che seguono l'attivazione della figura numero due creeranno il grafico nella figura 2. ⚠Se vogliamo tornare alla prima figura, basta scrivere `figure(1)`.

Se vogliamo chiudere una o più figure, scriveremo `close(2)` per chiudere la figura 2, oppure `close all` per chiudere tutte le figure.

Per finire lasciamo ai più curiosi di sperimentare questi comandi che permettono di modificare gli assi del grafico (vedere cosa succede al grafico dopo ogni istruzione relativa a **axis**).

```
» x=linspace(-2,2); y=sqrt(4-x.^2);
» plot(x,y)
» axis('square')
» axis('normal')
» axis('equal')
» axis([-2 2 1.5 2])
```

7.8 SALVARE I GRAFICI

Una volta fatto il grafico è importante salvarli su file (ad esempio su file .jpg o .pdf). Vediamo come fare dalla Command Window

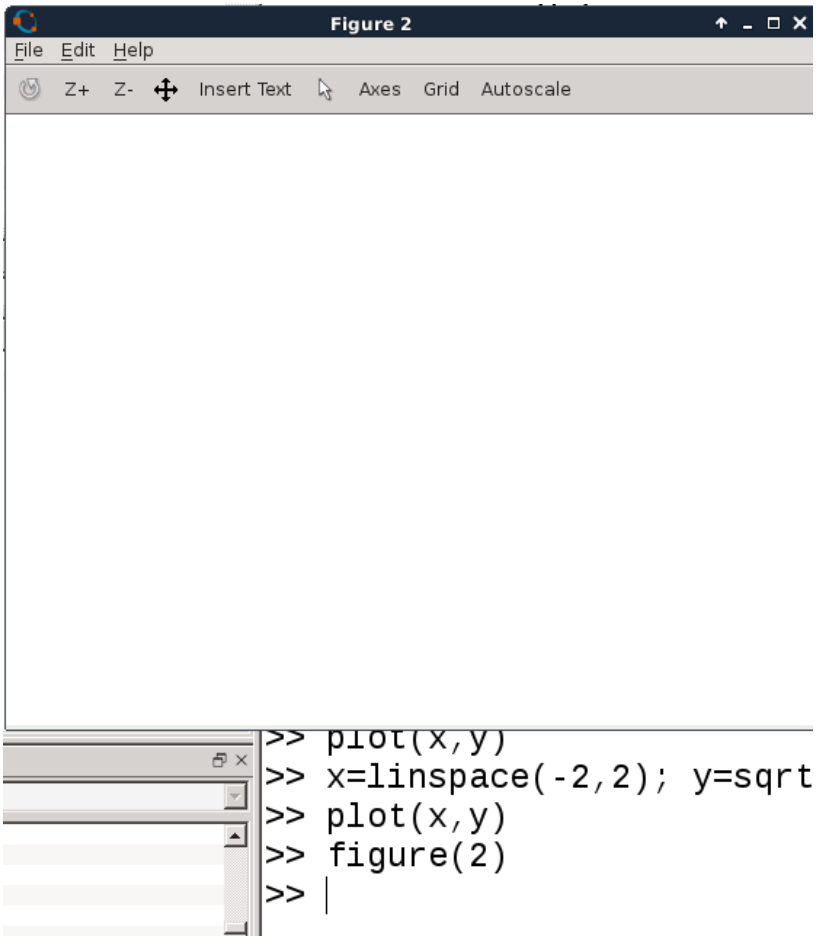


Figura 7.18: Finestra con una nuova figura da creare

o da uno script.

- Se abbiamo creato una sola figura e vogliamo salvarla con il nome `figura.jpg`, il comando è il seguente:
  print −djpg figura.jpg
- Se abbiamo creato più figure dobbiamo rendere attiva la figura che vogliamo salvare (e questo si fa o cliccando sulla figura o scrivendo `figure(1)` o `figure(2)` o... dipende dal nome della figura) e poi scrivere il comando di prima.

Se vogliamo salvare in .pdf basta sostituire `pdf` a `jpg` nelle istruzioni precedenti: **print** −dpdf figura.pdf

Perciò è corretto scrivere (in uno script o da Command Window)

```
figure(1)
plot(x1,y1)
title('titolo')
xlabel('etichetta')
print -Ppdf figura1.pdf
figure(2)
plot(x2,y2)
print -Ppdf figura2.pdf
```

Non è corretto da script scrivere

```
plot(x2,y2)
print -Ppdf figura1.pdf
print -Ppdf figura2.pdf
```

In questo caso avremo due file identici tra loro che corrispondono all'ultima figura.

Da Command Window si possono scrivere i due comandi uno di seguito all'altro se ⚠prima è stata attivata (cliccando sopra) la figura di cui voler fare il grafico!

8

METODI ITERATIVI PER ZERI DI FUNZIONE

7. A parole la gente sta con i perdenti, ma poi segue solo i vincenti. BATTITI COMUNQUE PER I PIÙ DEBOLI.
Kent M. Keith

ED ECCOCI ARRIVATI AL PRIMO GRANDE appuntamento con il Calcolo Numerico: essere capaci di risolvere equazioni non lineari, cercare cioè di appossimare gli zeri di una funzione scalare.

In Octave esiste una funzione predefinita, dal nome **fzero** che applica il metodo delle bisezioni per approssimare una radice della funzione data in input. Oltre alla funzione, in input occorre dare un vettore che abbia gli estremi di un intervallo in cui la funzione cambia segno, in modo da poter applicare il metodo delle bisezioni. Se vogliamo approssimare la radice della funzione $f(x) = -e^{-x} + 2\sin(x)$ nell'intervallo $[0,1]$, ($f(0)f(1) < 1$), possiamo operare in questo modo

```
» f=@(x) -exp(-x) +2*sin(x);
» x=fzero(f,[0,1])
x =  0.35733
```

Se vogliamo maggiori informazioni su quello che fa la function **fzero** possiamo studiare l'help in linea e applicarlo in questo modo

```
» [x, valorefx, info, output] = fzero(f, [0 1])
x =  0.35733
valorefx =    1.4433e-15
info =  1
output =

  scalar structure containing the fields:

    iterations =  7
    funcCount =  9
    bracketx =

       0.35733   0.35733
```

```
brackety =

   -1.1102e-16   1.4433e-15
```

In questo modo abbiamo non solo il valore della radice approssimata, ma anche il valore della funzione nella radice, quante iterazioni sono state effettuate, quante volte è stata valutata la funzione...

Noi però abbiamo studiato diversi metodi per approssimare zeri di funzione e vogliamo implementarli. 😌

8.1 Lo schema di punto fisso

Partiamo subito dall'algoritmo più semplice e che sarà da modello per tutti gli altri metodi iterativi, lo schema di punto fisso. 🏠

L'algoritmo di per sè è molto semplice:

- si parte da un'approssimazione iniziale x_0
- si applica l'algoritmo iterativo $x_{n+1} = g(x_n)$ dove g è la funzione di punto fisso

Se l'algoritmo converge, significa che per n che tende all'infinito x_{n+1} tende a ξ punto fisso della g.

Nella pratica ci fermeremo quando lo scarto $|x_{n+1} - x_n|$ tende a zero, cioè diventa minore di una tolleranza prefissata (dove per tolleranza prefissata si intende un valore molto piccolo come e 10^{-9}, 10^{-10}, 10^{-12} ...

Se invece l'algoritmo non converge, lo scarto non diventerà mai minore della tolleranza prefissata: allora bisognerà interrompere il procedimento quando ci si accorge di aver fatto un numero considerevole di passi (o iterazioni).

Se stiamo risolvendo a mano un esercizio dove bisogna applicare lo schema di punto fisso, noi lavoriamo nel modo seguente:

- consideriamo la funzione g e l'approssimazione iniziale x_0
- Prima iterazione: applichiamo il metodo per la prima volta ricavando il valore $x_1 = g(x_0)$
- Seconda iterazione: $x_2 = g(x_1)$
- Terza iterazione: $x_3 = g(x_2)$
- ...

Per capire quando dobbiamo arrestare l'algoritmo, ad ogni iterazioni calcoliamo lo scarto e lo confrontiamo con la tolleranza. Siccome stiamo lavorando con carta e penna, noi abbiamo

traccia di tutti i valori che approssimano la soluzione ad ogni iterazione: x_1, x_2, x_3, ... e gli scarti $d_1 = |x_1 - x_0|$, $d_2 = |x_2 - x_1|$, ...

Se vogliamo lavorare al computer possiamo fare qualcosa del genere (lavorando con un vettore delle approssimazioni ad ogni iterazione e un vettore degli scarti ad ogni iterazione) ma è bene salvare in vettori solo ciò che serve in modo da evitare di occupare troppo spazio in memoria. Gli scarti ci serviranno 😏‼️per fare i grafici di convergenza, e quindi è bene salvarli in un vettore. Le approssimazioni non servono tutte, quindi non conviene salvarle in un vettore.

Nella pratica, infatti, lavoreremo con poche variabili che hanno un preciso significato:

- `xold`: l'approssimazione della soluzione all'iterazione precedente
- `xnew`: l'approssimazione della soluzione all'iterazione corrente
- `iter`: l'iterazione che viene effettuata
- `vettscarti`: vettore degli scarti ad ogni iterazione per cui `vettscarti(i)` fornisce il valore dello scarto all'iterazione `i`-sima
- Quando partiamo abbiamo `x0` approssimazione iniziale che corrisponde, quindi, all'iterazione 0.
- Possiamo porre il valore di `x0` nella variabile `xold`.
- Applichiamo la prima volta l'algoritmo: avremo `xnew=g(xold)`. Inoltre `iter` deve valere 1.
- Possiamo quindi calcolare lo scarto all'iterazione 1 e salvarne il valore nella prima componente del vettore degli scarti.
- A questo punto dobbiamo passare all'iterazione 2, ma dobbiamo lavorare, abbiamo detto, solo con `xnew` e `xold`. Per fare questo trasferiamo il contenuto di `xnew` in `xold` perchè all'iterazione 2 a noi serve la formula $x_2 = g(x_1)$ ma x_1 è quanto abbiamo ora ricavato e il cui valore è in `xnew`. Imponendo `xold=xnew` noi perdiamo il contenuto che aveva prima `xold` (corrispondente a x_0) e vi poniamo il valore che è in `xnew` (cioè x_1). Perciò quando applichiamo nuovamente la formula `xnew=g(xold)` in `xnew` abbiamo effettivamente il valore che corrisponde all'iterazione 2, cioè x_2.
- Si va avanti in questo modo controllando ogni volta se lo scarto é minore della tolleranza oppure se stiamo superando il valore limite per le iterazioni.

L'algoritmo viene affidato ad un ciclo `while`.

```
        function [xnew,iter,vettscarti]=pfisso(g,x0,toll,itmax)
%function [xnew,iter,vettscarti]=pfisso(g,x0,toll,itmax)
%  significato delle variabili
%   iter : iterazione del metodo del punto fisso
%   itmax: numero massimo di iterazioni
%   toll : tolleranza prefissata per l'approssimazione del punto fisso
%   x0 : punto iniziale della successione
%   xold: approssimazione all'iterazione precedente
%   xnew: approssimazione all'iterazione corrente
%   vettscarti : vettore degli scarti
%   g:  funzione di cui si vuole calcolare un'approssimazione del punto fisso
%       data come function @
%
%   Esempio
%   x0=0.1; toll=1.e-12; itmax=100;
%   [xnew,iter,vettscarti]=pfisso(@cos,x0,toll,itmax)
%   xnew=0.7391
%   iter=70
%   vettscarti --- vettore di 70 componenti
%
%
vettscarti=zeros(itmax,1); %preallocazione del vettore degli scarti
scarto=2.0*toll; %  valore fittizio per entrare nel ciclo while
iter=0;  % prima di entrare nel ciclo  poniamo uguale a zero la
         % variabile iter in modo da poterla incrementare ad ogni
         % passo di una unita'
xold=x0;
while  scarto>=toll && iter<=itmax
         iter=iter+1; % stiamo applicando un passo del metodo iterativo
         xnew=g(xold); % approssimazione all'iterazione iter
         scarto=abs(xnew-xold); % scarto all'iterazione iter
         vettscarti(iter)=scarto; % conserviamo lo scarto nel vettore
                                  % degli scarti
         xold=xnew; %aggiornamento di xold con xnew in modo da poter
                    % applicare il passo successivo
end
vettscarti=vettscarti(1:iter);  % ora il vettore degli scarti ha
% la lunghezza delle  iterazioni effettivamente eseguite,
if (iter>itmax)
    disp('raggiunto il numero massimo di iterazioni')
    % messaggio di avvertimento per controllare
    % se la convergenza e' lenta o se il metodo sta divergendo
end
end
```

Oltre ai commenti già presenti nella function è importante sottolineare !!che il ciclo `while` viene effettuato fintantochè lo scarto è maggiore della tolleranza e (notare la e congiunzione) il numero delle iterazioni è minore del numero massimo di iterazioni consentite (e date in input). Fintantochè è vera la proposizione del ciclo, allora si va avanti con le iterazioni del metodo. Quando la proposizione che regge il ciclo `while` diventa falsa, allora si esce dal ciclo e ci si ferma. La proposizione diventa falsa o se lo scarto è diventato minore della tolleranza prefissata (cioè se siamo arrivati a convergenza e quindi il valore

`xnew` che viene dato in output approssima la soluzione del nostro problema) oppure se le iterazioni hanno superato il numero massimo consentito (e in questo caso vuol dire che il metodo sta divergendo oppure sta convergendo molto lentamente; se la convergenza è lenta occorrerà aumentare il numero massimo di iterazioni).

Questa function di punto fisso può essere usata per tutti i problemi in cui si vuole applicare lo schema di punto. Può essere richiamata più volte all'interno dello stesso script (se si vogliono risolvere più problemi); può essere richiamata con variabili di input che abbiano nomi diversi da quelli che abbiamo scritto nella function ma che hanno gli stessi significati;può essere usata, infine, come modello per tradurre altri algoritmi iterativi (Newton-Raphson, secante variabile, tangente fissa, tangente variabile...).

Vediamo ore diverse applicazioni della function di punto fisso e discutiamo diversi possibili errori (in modo da evitarli) 😱.

8.1.1 Applicazione dello schema di punto fisso

Un'applicazione molto semplice è applicare lo schema di punto fisso ad una funzione assegnata.

Prendiamo un esercizio tra i tanti che abbiamo (da fare con carta, penna e calcolatrice) e risolviamolo con uno script. La function che implementa l'algoritmo l'abbiamo già scritta. Non dobbiamo riscriverla ogni volta (a meno che non lo vogliamo fare per impararla bene). Importante è scrivere lo script nella stessa directory in cui si trova la function.

Esercizio Dato lo schema iterativo di punto fisso

$$x_{n+1} = \arctan(x_n) + \ln(x_n + 4), \quad n \geq 0$$

dimostrare esistenza e unicità del punto fisso nell'intervallo $I = [3,4]$; dire se il metodo converge nell'intervallo I e supponendo $x_0 = 3.9$ calcolare un'approssimazione del punto fisso eseguendo 4 iterazioni. Trovare inoltre l'ordine di convergenza, una stima del fattore di convergenza e una maggiorazione dell'errore.

Risoluzione 😱😎Non faremo 4 iterazioni (non dobbiamo lavorare con carta e penna) ma cercheremo un'approssimazione del punto fisso con una tolleranza, ad esempio, pari a 10^{-10}, ed entro un certo numero massimo di iterazioni uguale a 100. L'ultimo punto (la maggiorazione dell'errore) non lo risolviamo al calcolatore essendo una parte propriamente piu' analitica (e la lasciamo fare con carta e penna).

Scriviamo lo script e leggiamo con attenzione ‼️i commenti che vi riportiamo.

```
g=@(x) atan(x)+log(x+4);
a=3; b=4;
% grafico della funzione g insieme alla bisettrice
```

```
% y=x  (il grafico della bisettrice viene fatto
% prendendo gli estremi dell'intervallo e considerando
% i due vettori ascisse-ordinate uguali tra loro
% dal grafico si vede esistenza e unicita'
% (con carta e penna si fa diversamente)
ezplot(g,[a,b])
hold on
plot([a,b],[a,b])
hold off
% assegno i valori di tolleranza, numero massimo di iterazioni
% e il valore di x0
% ATTENZIONE: notare come e' stato scritto il valore  della tolleranza
% posso anche scrivere toll=10^(-10) o toll=10^-10 o toll=10.e-11
% NON POSSO SCRIVERE toll=10.e-10 perche' non avro' la tolleranza
% che e' stata prefissata
toll=1.e-10; itmax=100;
x0=3.9;
[xnew,iter,vettscarti]=pfisso(g,x0,toll,itmax)
% per calcolare l'ordine di convergenza controllo se
% il rapporto tra gli scarti e' lineare
% in tal caso il rapporto tra gli scarti e' la costante
% asintotica
M= vettscarti(2:iter)./vettscarti(1:iter-1)
% ATTENZIONE!!!!!!!!!!!!!
% abbiamo usato la divisione tra vettori in forma vettorizzata
% il tutto e' equivalente al ciclo for seguente
%  for i=1:iter-1
%     M(i)=vettscarti(i+1)/vettscarti(i)
%  end
%  In questo modo e' come se prendessi i vettori
%  V=vettscarti(2:iter) e  W=vettscarti(1:iter-1)
%  V ha le componenti dalla 2 alla iter di vettscarti
%  W ha le componenti dalla 1 alla iter-1 di vettscarti
%  infine faccio la divisione componente per componente
% dei due vettori V e W  (V./W)
% Il tutto viene invece fatto senza creare i due vettori in piu'
% e con un'unica riga
% !!!!!!!!!!!!!!!!!!
% grafico di convergenza
% apro un'altra figura per non perdere quella gia' fatta
figure(2)
semilogy([1:iter], vettscarti)
title('grafico di convergenza')
```

Quando eseguiamo lo script, i risultati sono in un formato che non ha le almeno sette cifre decimali richieste. Possiamo usare un **format** long per leggere più cifre decimali

```
» format long
» esercpfisso
xnew =  3.25425261412286
iter =  16
vettscarti =

   5.13343600375162e-01
   1.03302570828435e-01
```

```
....
....
   3.60799390364264e-10
   8.08659805784373e-11

M =

   0.201234749499048
   0.218843845470396
   0.222937731427380
   0.223862594695193
....
....
   0.224130325573928
   0.224130036630037
```

Dal rapporto degli scarti ci accorgiamo che l'ordine è lineare (come è di solito – ma non sempre !!– nello schema di punto fisso). E quindi una stima della costante asintotica è nell'ultima componente del vettore `M`. Volendo, per arricchire i nostri risultati, possiamo fare un grafico di convergenza. Le istruzioni, che possiamo scrivere all'interno dello script, le andiamo a leggere riprendendo, appunto, lo script (tornare indietro e leggere bene cosa abbiamo fatto). I grafici creati dallo script si trovano in Figura 8.1

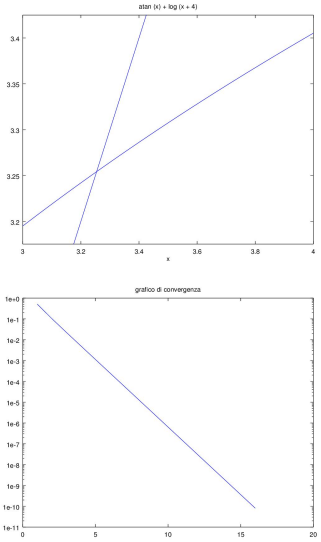


Figura 8.1: Risultati dello script sullo schema di punto fisso

Prendiamo ora spunto da un esercizio proposto ad un compito d'esame. La traccia (sfrondata di ciò che non è essenziale ai fini della programmazione) è la seguente:

Esercizio Si consideri l'equazione $2x^3 + x - 4 = 0$, che ammette un'unica soluzione nell'intervallo $[0,2]$. Si vuole risolvere il problema utilizzando uno schema di punto fisso. Infatti, possiamo considerare diverse funzioni di punto fisso, per le quali il punto fisso è soluzione dell'equazione proposta. Ad esempio, da $2x^3 + x - 4 = 0$ otteniamo $x = 4 - 2x^3$: la funzione di punto fisso é $g_1(x) = 4 - 2x^3$. Oppure $x = (\frac{4-x}{2})^{1/3}$, da cui $g_2(x) = (\frac{4-x}{2})^{1/3}$.

Si applichi quindi lo schema di punto fisso alle due funzioni g_1 e g_2 proposte, e si veda se entrambi gli schemi convergono e, in caso affermativo, a quale punto fisso. Si consideri una tolleranza $tol = 10^{-12}$ e si parta da $x_0 = 0$. A tale scopo si scriva uno script che:

1. definisca le funzioni $g_1(x)$ e $g_2(x)$ come function handle; [1]

2. definisca gli estremi dell'intervallo in cui approssimare il punto fisso (usando le variabili a e b);

3. definisca la variabile x_0;

[1] Attenzione all'uso delle parentesi in MATLAB® per scrivere correttamente le funzioni.

4. definisca il valore della tolleranza `toll` e del numero massimo di iterazioni `itmax` (prendendo 80 come numero massimo di iterazioni);

5. applichi, per ciascuna funzione di punto fisso, lo schema del punto fisso richiamando la function `pfisso` che abbia in input la funzione di punto fisso, l'approssimazione x_0, il valore della tolleranza e il numero massimo di iterazioni. In uscita, la function dia il valore approssimato della radice, il numero di iterazioni effettuate e il vettore con il valore dello scarto ad ogni iterazione. Lo schema iterativo del punto fisso va implementato fino a quando lo scarto tra due iterazioni successive non diventa minore della tolleranza prefissata o il numero di iterazioni non supera il numero massimo di iterazioni; si ricorda che la function `pfisso` deve essere generica e deve servire per implementare il metodo di punto fisso per una sola funzione di punto fisso;

6. per ogni schema di punto fisso, ci si faccia stampare l'approssimazione ottenuta e il numero di iterazioni effettuate;

7. per ogni schema di punto fisso, si controlli se è stato raggiunto o meno il numero massimo di iterazioni: se le iterazioni effettuate sono meno del numero massimo consentito, allora, usando il vettore degli scarti si dia una stima della costante asintotica dello schema, salvando i risultati in un opportuno vettore; se invece è stato raggiunto il numero massimo di iterazioni, si scriva un messaggio per dire che lo schema non sta convergendo;

8. si facciano quindi due grafici

   (a) una prima figura con le due funzioni di punto fisso e con la bisettrice del primo e terzo quadrante, nell'intervallo assegnato

   (b) una seconda figura con il grafico di convergenza in scala semilogaritmica sull'asse delle ordinate, in cui sulle ascisse si pone il valore delle iterazioni e sulle ordinate si pone il logaritmo in base 10 degli scarti, solo per gli schemi che risultano convergenti.

Le due figure vengano salvate in formato `pdf`.

Scriviamo il seguente script (⚠: si leggano con ATTENZIONE i commenti scritti)

```
g1 =@(x) 4 -2*x.^3;
g2 = @(x) ((4-x)/2).^(1/3);
% abbiamo creato due function handle per le due funzioni di punto fisso
% assegnate
tol=1.e-12;
itmax=80;
a=0;b=2;
x0=0;
```

```
% ora richiamo due volte di seguito la stessa function di punto fisso
% dando nomi diversi alle variabili di output in modo da conservarne
% i risultati
% in input cambia solo la function handle
% ATTENZIONE: la function di punto fisso e' una sola e viene richiamata
% due volte !!!!!!!!!!!!!!!!
[xnew1,iter1,vettscarti1]=pfisso(g1,x0,tol,itmax);
[xnew2,iter2,vettscarti2]=pfisso(g2,x0,tol,itmax);
disp([xnew1,iter1])
disp([xnew2,iter2])
% senza vedere i risultati controllo se le iterazioni effettuate sono
% maggiori del numero massimo
% se iter1 < itmax vuol dire che sto andando a convergenza e ipotizzando
% convergenza lineare faccio il rapporto tra gli scarti
% se iter1 > itmax la convergenza e' lenta e mostro un messaggio di
% avvertimento
if iter1<itmax
M1=vettscarti1(2:end)./vettscarti1(1:end-1);
else
 disp('primo schema non sta convergendo')
end
% ripeto la stessa cosa di prima per il secondo schema
if iter2<itmax
M2=vettscarti2(2:end)./vettscarti2(1:end-1);
else
 disp('secondo schema non sta convergendo')
 end
% prima figura: le due funzioni di punto fisso e la bisettrice
figure(1)
ezplot(g1,[a,b])
hold on
ezplot(g2,[a,b])
plot([a,b],[a,b])
hold off
% seconda figura: faccio i grafici di convergenza per lo/gli schemi convergenti
% percio' rifaccio il controllo sulle iterazioni
if iter1<itmax  % se iter1<itmax faccio il grafico (se no, non lo faccio)
figure(2)
semilogy([1:iter1],vettscarti1)
hold on
end
if iter2<itmax  % se iter2 < itmax faccio il grafico (se no, non lo faccio)
figure(2)
 semilogy([1:iter2],vettscarti2)
 hold off
end
```

8.2 COME IMPLEMENTARE GLI ALTRI METODI

Siamo ora in grado di scrivere function per implementare gli schemi di Newton-Raphson, della secante variabile, della tangente fissa, ... tutti schemi iterativi in cui ad ogni passo va applicata una precisa formula.

Cosa cambia?
Confrontiamo le formule:

Punto fisso	Tangente fissa	Secante fissa	Newton-Raphson
	`c= df(x0)`	`c= (f(x1)-f(x0))/(x1-x0)`	
`xnew=g(xold)`	`xnew=xold-f(xold)/c`	`xnew=xold -f(xold)/c`	`xnew=xold - f(xold)/df(xold)`

Osserviamo che per la tangente fissa e per la secante fissa, prima di scrivere la formula del metodo abbiamo introdotto una variabile `c` che, una volta definita, non cambia più il suo valore. Questa variabile `c` può essere data in input alla function dello schema insieme agli altri parametri di input oppure può essere definita all'interno della function dello schema prima di entrare nel ciclo while.

Per lo schema di Newton-Raphson, invece, ad ogni iterazione abbiamo bisogno del valore della funzione di cui si vuole approssimare la radice e della sua derivata prima: ciò significa che oltre alla funzione f è necessario dare in input alla function anche la derivata prima della f (sempre come function handle).

Quindi per questi schemi, possiamo impostare le function in questo modo

- Per lo schema della tangente fissa

```
function [xnew, iter, vettscarti]=tangentefissa(f,x0,c,toll,itmax)
```

In tal caso si passa tra i parametri di input `c=f(x0)`. Oppure

```
function [xnew, iter, vettscarti]=tangentefissa(f,x0,toll,itmax)
```

In questo caso si definisce `c` all'interno della function, prima del ciclo while.

- Per lo schema della secante fissa

```
function [xnew, iter, vettscarti]=tangentefissa(f,x0,c,toll,itmax)
```

In tal caso si passa tra i parametri di input `c=(f(x1)-f(x0))/(x1-x0)`, dove `x1` è un valore iniziale (per lo schema della secante fissa e della secante variabile abbiamo bisogno di due valori iniziali per poter far partire lo schema). Oppure

```
function [xnew, iter, vettscarti]=tangentefissa(f,x0,x1,toll,itmax)
```

In questo caso si definisce `c` all'interno della function, prima del ciclo while, ma occorre passare la variabile `x1` tra le variabili di input.

- Per lo schema di Newton-Raphson

```
function [xnew, iter, vettscarti]=newton(f,df,x0,toll,itmax)
```

Qui `df` è la funzione derivata prima di f che è stata definita (facendo i calcoli con carta e penna) a seconda della funzione f. Occorre definire due function handle, la funzione f e la sua derivata, da passare in input alla function di Newton-Raphson.

Nel corso di Calcolo Numerico, non riusciamo a fare il simbolico, quindi la derivata la calcoliamo con carta e penna, senza usare function di differenziazione automatica.

A parte queste indicazioni, ciò che cambia rispetto allo schema del punto fisso è l'algoritmo del metodo da implementare (che abbiamo visto prima). Tutto il resto rimane invariato:

```
        function [xnew,iter,vettscarti]=nomefunction(variabili di input)
%  significato delle variabili
%   iter : iterazione del metodo del punto fisso
%   itmax: numero massimo di iterazioni
%   toll : tolleranza prefissata per l'approssimazione del punto fisso
%   x0 : punto iniziale della successione
%   xold: approssimazione all'iterazione precedente
%   xnew: approssimazione all'iterazione corrente
%   vettscarti : vettore degli scarti
%   commenti alle altre variabili
%
vettscarti=zeros(itmax,1); %preallocazione del vettore degli scarti
scarto=2.0*toll; %  valore fittizio per entrare nel ciclo while
iter=0;  % prima di entrare nel ciclo  poniamo uguale a zero la
         % variabile iter in modo da poterla incrementare ad ogni
         % passo di una unita'
xold=x0;
%%% qui eventuali istruzioni sulla variabile c (se occorre)
while  scarto>=toll && iter<=itmax
         iter=iter+1; % stiamo applicando un passo del metodo iterativo
         xnew= APPLICARE LA FORMULA % approssimazione all'iterazione iter
         scarto=abs(xnew-xold); % scarto all'iterazione iter
         vettscarti(iter)=scarto; % conserviamo lo scarto nel vettore
                                  % degli scarti
         xold=xnew; %aggiornamento di xold con xnew in modo da poter
                    % applicare il passo successivo
end
vettscarti=vettscarti(1:iter);  % ora il vettore degli scarti ha
% la lunghezza delle  iterazioni effettivamente eseguite,
if (iter>itmax)
    disp('raggiunto il numero massimo di iterazioni')
    % messaggio di avvertimento per controllare
    % se la convergenza e' lenta o se il metodo sta divergendo
end
end
```

Le modifiche sono minime (e le lasciamo come esercizio).

Per lo schema della secante variabile (o Regula-Falsi), bisogna, invece, pensarci un attimo di più 😔ma le cose non sono affatto complicate 😱.

Lo schema infatti ora è

$$x_{n+1} = x_n - f(x_n)/c, \text{ dove } c = \frac{f(x_n) - f(x_{n-1}}{x_n - x_{n-1}}$$

Dobbiamo sempre partire da due valori iniziali x_0 e x_1 e poi ad ogni iterazione dobbiamo lavorare con tre variabili: x_{n-1}, x_n e x_{n+1}.

Perciò, tra le variabili di input, occorre dare x_1 oltre a x_0. Invece quando si implementa l'algoritmo, oltre a `xold` e `xnew`, (che hanno sempre il significato di x_n e x_{n+1}) ci converrà introdurre una terza variabile che corrisponde a x_{n-1} e che possiamo chiamare `xoldold` o `xoldmeno` (o un altro nome che però ci deve

ricordare a cosa corrisponde – non chiamiamola `pincopallino` o `boh` 😉).

Conviene quindi partire assegnando il valore di `x0` alla variabile `xoldold`, e il valore di `x1` alla variabile `xold`. Prima di applicare la formula iterativa (all'interno del ciclo while) conviene fare il rapporto incrementale (variabile `c`) stando bene attenti all'uso delle parentesi ‼️. Poi si applica la formula. Si aggiorna lo scarto, il vettore degli scarti, (non cambia nulla rispetto agli altri schemi). Quindi si deve fare l'aggiornamento delle variabili `xold` e `xoldold`. E qui bisogna stare attenti a non perdere informazioni!!! Bisogna cioè ragionare su come vanno aggiornate queste variabili.

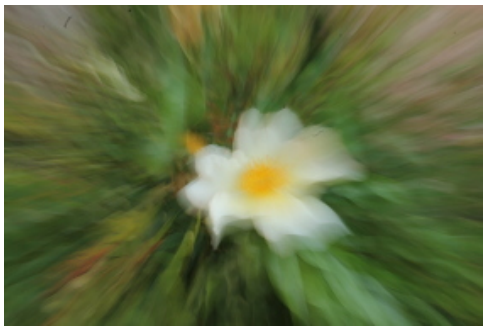
Lo schema della secante variabile può essere ottimizzato lavorando su due variabili soltanto, ma dal momento che stiamo imparando a programmare, lasciamo perdere questi dettagli.

x0	x1	x2
xoldold	xold	xnew
↙	↙	
x1	x2	x3
xoldold	xold	xnew
↙	↙	
x3	x4	x5
xoldold	xold	xnew
…	…	…

Dal prospetto fatto vediamo che ad ogni passo il valore di `xnew` dovrà passare a `xold` e il valore di `xold` dovrà passare a `xoldold`. Quindi l'istruzione che avevamo negli altri schemi xold=xnew continua a valere, ma si deve considerare anche xoldold=xold. Passiamo `xnew` a `xold` (xold=xnew) ma dobbiamo passare anche `xold` a `xoldold` (xoldold=xold).

E qui poniamo l'amletico quesito: in quale ordine vanno poste le due istruzioni senza perdere informazioni? 🙄

La risposta è lasciata al lettore: si ragioni un po' considerando che una volta effettuata l'operazione di assegnazione (=) il contenuto della variabile aggiornata viene appunto aggiornato e non rimane traccia di quello che era prima; si facciano delle prove (assegnando dei valori numerici alle tre variabili `xnew`, `xold` e `xoldold`) e vedendo cosa si deve fare per aggiornare correttamente `xold` e `xoldold`). La risposta non è difficile (tranquilli 😪😩😊).



9

Interpolazione e approssimazione di dati

8. Ciò che hai impiegato anni a costruire può crollare in un istante. SCEGLI COMUNQUE DI COSTRUIRE.
Kent M. Keith

Il problema di interpolare e approssimare dati può essere risolto in vari modi. Qui vedremo soltanto interpolazione e approssimazione di dati usando polinomi e utilizzando function predefinite.

Le function che utilizzeremo saranno principalmente due: **polyfit** e **polyval**.

9.1 Interpolazione

Supponiamo di avere delle misure (t_i, v_i) $i = 1, 2, \ldots, n$ in cui t_i rappresentano i secondi e v_i le velocità in metri al secondo, di un razzo che è appena partito dal suolo.

Vogliamo interpolare i dati.

Per prima cosa, andremo a costruire due vettori, uno per le ascisse, uno per le ordinate: possiamo chiamarli `t` e `v` oppure `x` e `y`...

Poi andiamo a vedere la lunghezza di questi vettori (se abbiamo pochi elementi lo vediamo subito ma se dobbiamo leggere dati da tabelle, conviene usare la function predefinita **length** che ci dice la lunghezza dei vettori). Sappiamo che se abbiamo $n+1$ coppie di punti da interpolare, il polinomio di interpolazione sarà di grado n. Se abbiamo n coppie di punti il polinomio di interpolazione sarà di grado $n-1$.

Quindi applichiamo la function **polyfit** con variabili di input date dai due vettori e dal grado del polinomio di interpolazione. In output avremo un vettore che ha i coefficienti del polinomio di interpolazione, dal coefficiente di grado più alto a quello di grado più basso (ordine decrescente).

Facciamo subito un esempio dalla Command Window. I dati da interpolare siano

t	v
0	0
10	225.02
14	365.76
18	518.24
23.5	602.89
30	905.12

```
» x=[0 10 14 18 23.5 30]; y=[0 225.02  365.76 518.24 602.89 905.12];
» n=length(x)
n =  6
» m=n-1;
» p=polyfit(x,y,m)
p =

   1.1074e-03  -7.9956e-02   2.0082e+00  -2.0297e+01   9.3535e+01   1.1773e-11
```

Il vettore `p` ha sei componenti perchè il polinomio di interpolazione è di grado 5.

Se ora vogliamo valutare il polinomio in un punto o in più punti, ci viene in aiuto la function **polyval**. Se, ad esempio vogliamo valutare la velocità al tempo $t = 15s$ usando il polinomio di interpolazione, basta fare

```
» polyval(p,15)
ans =  407.02
```

oppure

```
» t=15; yt=polyval(p,t);
» yt
yt =  407.02
```

Tutto ciò può essere utile per fare il grafico del polinomio di interpolazione. Creiamo un vettore delle ascisse nell'intervallo in cui vogliamo fare il grafico; valutiamo il polinomio in questi punti creando quindi un altro vettore e poi facciamo il grafico dei vettori ascisse-ordinate che abbiamo. Supponiamo di voler fare il grafico nell'intervallo individuato dalle ascisse dei nodi di interpolazione. In questo caso le ascisse sono messe in ordine crescente, quindi possiamo fare qualcosa del genere:

```
» xval=linspace(x(1),x(n));
» yval=polyval(p,xval);
» plot(x,y,'o','Markersize', 6, xval, yval, 'linewidth'
```

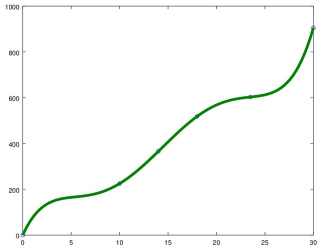


Figura 9.1: Interpolare dati

⚠!!ATTENZIONE: Le ascisse di interpolazione possono non essere in ordine crescente. Possiamo quindi anche scrivere xval=**linspace**(**min**(x),**max**(x)) andando a prendere gli estremi dell'intervallo in cui fare il grafico tra il più piccolo e il più grande

dei valori delle ascisse, usando le function predefinite **min** e **max**.

⚠!!ATTENZIONE: se non abbiamo definito una variabile per la lunghezze del vettore e vogliamo considerare l'ultima sua componente possiamo anche scrivere x(**end**). **end** dà risultati solo se usato all'interno di un vettore o di una matrice.

9.1.1 Interpolazione di funzioni

Consideriamo un problema come quello di interpolare una funzione assegnata in determinati punti, semmai aumentando il numero di punti e, di conseguenza, il grado del polinomio di interpolazione.

Vediamo questo esercizio.

Esercizio Data la funzione $f(x) = \cos(x-2) - e^{(1-x)}$, si vogliono costruire i polinomi di interpolazione, rispettivamente di grado $n = 3, 5, 10$, nell'intervallo $[0, 8]$. In particolare, si vogliono fare dei grafici (della funzione f e dei e polinomi di interpolazione), e si vuole vedere l'errore relativo che si commette se si approssima $f(3)$ con i tre polinomi di interpolazione.

A tale scopo si scriva uno script MATLAB® che:

1. definisca la funzione f come function handle;
2. introduca due variabili `a` e `b` per gli estremi dell'intervallo di interpolazione;
3. costruisca un vettore `nvett` che contenga il grado dei tre polinomi di interpolazione;
4. costruisca il vettore `xvett` di punti equidistanti tra `a` e `b` (di 50 o 100 componenti) che servirà per valutare i polinomi di interpolazione e fare i grafici;
5. costruisca una variabile `c` dove salvare il valore $x = 3$ in cui andare a valutare i polinomi di interpolazione (per confrontare poi con il valore $f(3)$);
6. costruisca il vettore `polc` di 3 componenti nulle in cui andare poi a salvare il valore dei polinomi in `c`;
7. Fare quindi un ciclo (`for`) per poter ricorsivamente costruire e valutare i polinomi di interpolazione a seconda del grado richiesto;
8. All'interno del ciclo:
   - salvare in `n` il grado del polinomio che si deve costruire, usando in modo opportuno il vettore `nvett`;
   - creare il vettore delle ascisse dei nodi di interpolazione per il valore di `n` che si sta utilizzando;

- creare il corrispondente vettore delle ordinate dei nodi di interpolazione;
- costruire il vettore con i coefficienti del polinomio di interpolazione di grado `n`;
- valutare il polinomio nel vettore `xvett` salvando i risultati nel vettore `yvett`;
- valutare il polinomio nella variabile `c` salvando il risultato nella opportuna componente del vettore `polc`;
- aprire la figura (che cambierà a seconda del polimonio creato, se è il primo, il secondo o il terzo);
- fare il grafico della funzione f e sovrascrivere il grafico del polinomio di interpolazione (si faccia sia il grafico – per punti – dei nodi di interpolazione, sia il grafico – per linee – del polinomio di interpolazione, utilizzando i valori ottenuti con `xvett`);

9. una volta usciti dal ciclo si costruisca un vettore con l'errore relativo rispetto al punto `c` (le componenti devono dare il valore $\frac{|f(c)-p_n(c)|}{|f(c)|}$);
10. si stampi sulla Command Window questo vettore.

Per risolvere questo esercizio mettiamo in pratica diverse conoscenze che abbiamo già acquisito. Leggiamo quindi con MOLTA ATTENZIONE 😎⚠tutti i commenti scritti.

```
close all
% chiudiamo eventuali figure, se ce ne sono
clear   % cancelliamo variabili
f= @(x) cos(x-2) -exp(1-x); % la funzione da interpolare
a=0; b=8;  % estremi dell'intervallo
nvett=[3,5,10];  % gradi dei polinomi di interpolazione
% vogliamo costruire tre polinomi di interpolazione
xvett=linspace(a,b);  % vettore delle ascisse in cui andare a
% valutare i polinomi
c=3;  % valore che ci servira' per valutare i polinomi
polc=zeros(3,1); %inizializzazione del vettore che conservera'
% i valori dei tre polinomi di interpolazione valutati in c
for i=1:3  % questo ciclo for ci permette di costruire i tre polinomi
           % di interpolazione
    n=nvett(i); % fissato i, prendiamo la componente nvett(i)
                % in modo da costruire il polinomio di interpolazione
                % con quel grado (n=3, 5, 10, a seconda che i=1,2,3)
    ascisse=linspace(a,b,n+1); % vettore delle ascisse da interpolare
    ordinate=f(ascisse); % vettore delle ordinate da interpolare
    % considero n+1 punti perche' il grado del polinomio deve
    % essere n
    p=polyfit(ascisse,ordinate,n); % vettore con le componenti
    % del polinomio di interpolazione di grado n
    yvett=polyval(p,xvett);  % valutazione del polinomio in xvett
    polc(i)=polyval(p,c); % valutazione del polinomio in c
    figure(i)  % apriamo la figura i (sara' 1,2,3)
    ezplot(f,[a,b]) % grafico della funzione da interpolare
```

```
    hold on
    plot(ascisse,ordinate,'co', xvett,yvett,'k')
    % grafico del polinomio di interpolazione
    % insieme ai nodi di interpolazione
    hold off
    title(['Interpolazione di grado ' num2str(n)])
end
err=abs(f(c)-polc)./abs(f(c)); % vettore con gli errori rela
% di interpolazione nel punto c, al variare dei tre polinomi .....
disp('errore relativo')
disp(err)
```

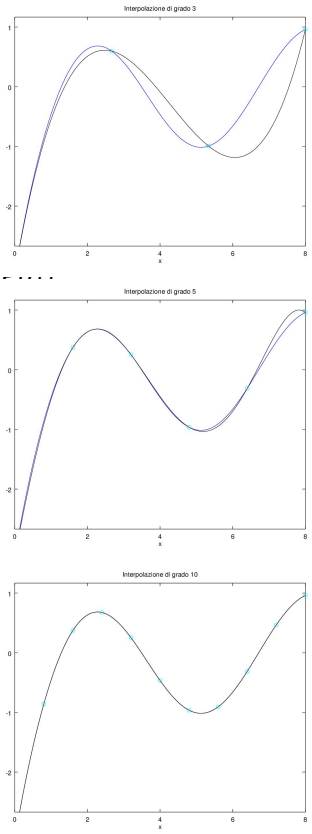


Figura 9.2: Interpolare una funzione

Quando eseguimo lo script otteniamo i seguenti risultati

```
errore relativo
   2.5261e-01
   8.2728e-03
   1.8718e-05
```

E i grafici sono quelli in Figura 9.2.

9.2 APPROSSIMAZIONE

La function **polyfit** ci serve anche per approssimare dati. Infatti l'algoritmo che sta alla base di questa function applica un procedimento di approssimazione ai minimi quadrati con grado che va da 0 (funzione costante) fino al grado di approssimazione per cui approssimare vuol dire interpolare (quanto abbiamo fatto prima).

Perciò se applichiamo la function **polyfit** a $n+1$ coppie di punti e diamo in input le coppie di punti e $m = 1$, otteniamo la retta, se mettiamo $m = 2$ abbiamo un polinomio di approssimazione di grado 2, e così via.

Facciamo subito un esempio, prendendo i dati di tempo e velocità usati prima, per ottenere la retta di approssimazione ai minimi quadrati.

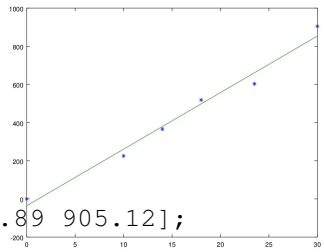


Figura 9.3: Approssimare dati

```
» x=[0 10 14 18 23.5 30]; y=[0 225.02  365.76 518.24 602.89 905.12];
» coefretta=polyfit(x,y,1)
coefretta =

   29.638  -35.564

» xval=linspace(x(1),x(end)); yval=polyval(coefretta,xval);
» plot(x,y,'*',xval,yval)
```

Ricordando che i coefficienti sono dati in ordine decrescente, volendo scrivere la retta come $y = a_0 + a_1 x$ si ha $a_1 = 26.638$, $a_0 = -35.564$, cioè $y = -35.563 + 26.638x$. Quindi vediamo la relazione `coefretta=[coefretta(1), coefretta(2)]`= $[a_1, a_0]$.

9.3 CARICARE DATI

A volte i dati sono tanti e non conviene scriverli all'interno di uno script. Oppure li abbiamo già in un file. Conviene quindi cercare di leggerli e assegnarli a delle variabili senza doverli riscrivere.

Supponiamo di avere questo file (o di scriverlo noi con un editor di testo che salvi in formato ASCII, cioè senza formattare caratteri).

Ad esempio abbiamo scritto in un file dal nome `dati.dat` questi dati che corrispondono alle ascisse-ordinate da interpolare o da approssimare.

```
1 0.5
2 1.4
3 3.7
4.5 8.1
5 10.4
6 13.7
7.5 21.9
8 24.6
9.1  32.4
10  40.3
```

Abbiamo i dati in due colonne (prima colonna: ascisse, seconda colonna: ordinate), e vogliamo leggerli in Octave. Come fare? Carichiamo i dati in una matrice usando la function predefinita **load** e poi ci prendiamo le due colonne. Vediamo come.

⚠!!Il nome del file da dare in input alla function **load** va scritto tra apici, come stringa di caratteri oppure deve essere già una variabile di caratteri (una stringa definita prima).

😔Per completezza, introduciamo anche il comando **save** che permette di salvare su file variabili che si trovano sulla Command Window. Tra le tante possibilità che abbiamo ricordiamo quella che ci permette di salvare in formato ASCII. Se abbiamo una matrice `A` e vogliamo salvarla nel file `matriceA.dat`, scriviamo `save -ascii matriceA.dat A`.

```
» A=load('dati.dat')
A =

    1.00000    0.50000
    2.00000    1.40000
    3.00000    3.70000
    4.50000    8.10000
    5.00000   10.40000
    6.00000   13.70000
    7.50000   21.90000
    8.00000   24.60000
    9.10000   32.40000
   10.00000   40.30000

» x=A(:,1); y=A(:,2);
```

Non abbiamo messo il punto e virgola dopo l'istruzione di caricamento dati per vedere cosa accade.

Salviamo poi le colonne in due variabili `x` e `y` in modo da poter poi interpolare o approssimare i dati.

Se scriviamo il file non su colonne ma su righe dobbiamo stare attenti a prendere le righe, altrimenti facciamo errore. Sia dato (o scritto) il file `datiriga.dat`.

```
1 2 3 4.5 5 6 7.5 8 9.1 10
0.5 1.4 3.7 8.1 10.4 13.7 21.9 24.6 32.4 40.3
```

Ora i dati sono su due righe e hanno il significato di: prima riga - ascisse, seconda riga - ordinate. Quindi dobbiamo operare nel modo seguente

```
» B=load('datiriga.dat')
B =

 Columns 1 through 7:

    1.00000    2.00000    3.00000    4.50000    5.00000    6.00000    7.50000
    0.50000    1.40000    3.70000    8.10000   10.40000   13.70000   21.90000

 Columns 8 through 10:

    8.00000    9.10000   10.00000
   24.60000   32.40000   40.30000
» x=B(1,:); y=B(2,:);
```

Abbiamo preso le due righe (quindi ora abbiamo due vettori riga). Se facessimo al contrario, (come nel caso precedente) prenderemmo solo due elementi:

```
» x=B(:,1)
x =

   1.00000
   0.50000

» y=B(:,2)
y =

   2.0000
   1.4000
```

9.4 Approssimazione secondo modelli potenza ed esponenziale

La function **polyfit** torna utile se vogliamo approssimare i dati secondo un modello potenza o esponenziale o del tipo $y = a_0 + a_1 x^m$. Dalla teoria abbiamo visto che questi sono tutti casi che si possono ricondurre alla retta di approssimazione ai minimi quadrati facendo opportune trasformazioni (in alcuni casi passando ai logaritmi, in altri casi no).

⚠Un errore comune è pensare che un modello del tipo $y = a_0 + a_1 x^m$ (con m intero) vada trasformato usando i logaritmi!!! In tal caso, invece, basta porre $X = x^m$ per avere $y = a_0 + a_1 X$.

Vediamo uno di questi casi con un esercizio.

Esercizio

Un punto materiale si muove lungo una certa traiettoria ed è possibile misurare, per diversi istanti di tempo, la distanza percorsa. I risultati ottenuti sono in tabella:

t [s]	1	2	3	4.5	5	6	7.5	8	9.1	10
s [m]	2.1	5.5	11.7	24.3	32.1	45.4	69.9	82.6	98.4	125.8

Dalle misure ottenute:

- assumendo che il moto sia rettilineo uniforme, si vuole trovare la retta di approssimazione nel senso dei minimi quadrati, $s(t) = s_0 + vt$ che approssima i dati (dove s_0 rappresenta la distanza al tempo $t = 0$ e v la velocità);

- assumendo che il moto del punto materiale sia uniformemente accelerato con velocità iniziale nulla, si vuole determinare, nel senso dei minimi quadrati, la funzione $d(t) = d_0 + at^2$ dove a è l'accelerazione del punto materiale e d_0 la distanza al tempo $t = 0$.

Delle due funzioni si vuole fare anche un grafico.

Per risolvere il problema, si scriva innanzitutto un file con i dati e lo si salvi con il nome `dati1.dat`. Si scriva poi uno script strutturato nel modo seguente:

1. carichi le coppie di dati sperimentali (che devono essere caricati da file e non devono essere scritti a mano all'interno dello script!)

2. utilizzando function proprie di MATLAB® si trovino i coefficienti s_0 e v della retta di approssimazione $s(t) = s_0 + v_0 t$;

3. utilizzando function proprie di MATLAB® si trovino i coefficienti d_0 e a della funzione di approssimazione $d(t) = d_0 + at^2$;

4. per confrontare le due approssimazioni, si calcoli la somma dei quadrati degli scarti rispetto alle due funzioni, salvando i risultati rispettivamente nelle variabili `scartoretta` e `scartocurva`;

5. si faccia poi un unico grafico che, utilizzando al più 100 punti equidistanti presi tra il più piccolo e il più grande degli istanti temporali a disposizione, mostri le due funzioni $s(t)$ e $d(t)$ ottenute, insieme alle misure sperimentali (le due funzioni siano mostrate con linee, le misure sperimentali con punti).

Per risolvere questo esercizio faremo uso di una function predefinita dal nome **sum** che, applicata ad un vettore, fa la somma delle componenti del vettore.

Il file di dati che abbiamo scritto usando un editor di testo è mostrato in Figura 9.4

Figura 9.4: Tabella di dati in un file

Lo script è il seguente (leggiamo sempre con molta ATTENZIONE tutti i commenti 😎)

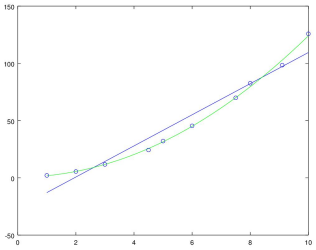


Figura 9.5: Grafico generato dallo script di approssimazione

```
clear
close all
A=load('dati1.dat');
t=A(:,1);
s=A(:,2);
p=polyfit(t,s,1);
% i coefficienti di p corrispondono a p=[p(1), p(2)]=[v s0]
s0=p(2);
v=p(1);
T=t.^2;
% prendo le ascisse dei dati e le elevo al quadrato in modo da
% poter riutilizzare il modello dei minimi quadrati
q=polyfit(T,s,1);
% ora q=[q(1) q(2)]=[a d0]
d0=q(2);
a=q(1);
tt=linspace(t(1),t(end));
% creo in tt un vettore di 100 punti equidistanti tra il primo e l'ultimo
% istante temporale (i valori sono in ordine crescente)
ss=polyval(p,tt);
% valuto la retta di approssimazione nei punti tt
d=@(t) d0+a*t.^2;
% creo invece una function handle per esprimere il modello y=d0+at^2
% non posso usare la polyval sul vettore q perche' avrei una retta
% eventualmente dovrei creare il vettore q1=[q(1) 0 q(2)] in
% quanto il coefficiente di x^1 e' nullo
dd=d(tt);
% valuto il modello y=d0+at^2 nei punti tt
plot(tt,ss,'b',tt,dd,'g',t,s,'o');
% creo il grafico con i due modelli e i punti assegnati
puntiretta=polyval(p,t);
% valuto nei punti t assegnati, quanto vale la retta di approssimazione
scartoretta=sum((puntiretta-s).^2)
% cosi' posso calcolare la somma dei quadrati degli scarti
```

```
% faccio la differenza tra i due vettori, componente per componente,
% elevo al quadrato, componente per componente,
% infine tramite la function predefinita sum faccio la somma
% delle componenti del vettore risultante
punticurva=d(t);
scartocurva=sum((punticurva-s).^2)
% applico il procedimento di prima per calcolare la somma del quadrato
% degli scarti per il modello y=d0+at^2
```

Eseguendo lo script abbiamo

```
» eserappr
scartoretta =  843.84
scartocurva =  33.934
```

Vediamo un ultimo esempio considerando i dati della tabella precedente (per semplicità) pero approssimarli tramite un modello potenza del tipo $s = kt^m$. In questo caso, per ricondurci al modello di approssimazione ai minimi quadrati, dobbiamo ricordarci cosa fare (cioè la TEORIA !!). Applichiamo quindi i logaritmi (naturali o in base 10) per avere $\log s = \log k + m \log t$. E poi dobbiamo ricordarci di tornare indietro. Le istruzioni da scrivere sono le seguenti.

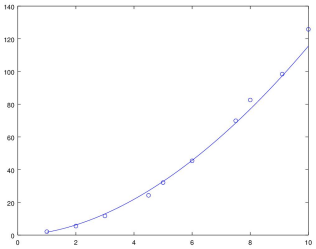


Figura 9.6: Grafico generato dallo script sul modello potenza

```
clear
close all
A=load('dati1.dat');
t=A(:,1);
s=A(:,2);
x=log10(t);
y=log10(s);
% siamo passati ai logaritmi in base 10 in modo da poter
% ricavare la retta di approssimazione ai minimi quadrati
p=polyfit(x,y,1);
% i coefficienti di p corrispondono a p=[p(1), p(2)]=[m, log10(k)]]
k=10^p(2);
m=p(1);
% Ora creiamo la funzione handle del modello ottenuto.
% I parametri k e m sono stati creati, quindi esistono come valori
f=@(t) k.*t.^m;
tt=linspace(t(1),t(end));
% creo in tt un vettore di 100 punti equidistanti tra il primo e l'ultimo
% istante temporale (i valori sono in ordine crescente)
ss=f(tt);
% valuto il modello  nei punti tt
plot(tt,ss,'b',t,s,'o');
% creo il grafico con i due modelli e i punti assegnati
% per ottenere la somma dei quadrati degli scarti
punticurva=f(t);
scarto=sum((punticurva-s).^2)
% applico il procedimento  per calcolare la somma del quadrato
% degli scarti sul modello ottenuto
```

```
» eserappr2
scarto =  150.56
```

10

Matrici e vettori (parte 2)

9. Le persone hanno bisogno di aiuto ma potrebbero attaccarti se gli offri una mano. AIUTALE COMUNQUE.
Kent M. Keith

Il punto di forza dell'ambiente di programmazione che stiamo imparando ad usare è dato dalle matrici.

Vediamo ora come risolvere problemi di algebra lineare, partendo dal costruire matrici e vettori con particolaari proprietà per poi risolvere sistemi lineari o trovare autovalori e autovettori di matrici.

😔😒Vedremo alcuni elementi di base.

10.1 Matrici e vettori speciali

Per vedere ora come creare matrici e vettori speciali, lavoriamo direttamente sulla Command Window: i comandi che diamo dando in input il numero di righe e di colonne creano matrici e, nel caso particolare, in cui una delle due dimensioni vale 1, ci riconduciamo a vettori. Vediamo perciò in generale lasciando come esercizio di vedere cosa bisogna fare per avere i corrispondenti vettori riga o colonna.

```
» n=2; m=3;
```

- Una matrice con elementi tutti uguali a uno

```
=ones(n,m)
A =

    1    1    1
    1    1    1
```

- matrice identità

```
» A=eye(n)
```

```
A =

Diagonal Matrix

   1   0
   0   1
```

Possiamo anche scrivere A=**eye**(m,n) per avere matrici rettangolari con elementi della diagonale principale uguali a 1.

- Creare una matrice diagonale o estrarre la diagonale da una matrice

```
» vettorediagonale=[1 2 3];
» A=diag(vettorediagonale)
A =

Diagonal Matrix

   1   0   0
   0   2   0
   0   0   3
» v=diag(A)
v =

   1
   2
   3
```

Se poi gli elementi diagonali diversi da zero vogliamo metterli sopra o sotto la diagonale principale, e precisamente sulla k-sima sopra o sotto diagonale, aggiungiamo il valore k come dato di input

```
» A=diag(v,2)
A =

   0   0   1   0   0
   0   0   0   2   0
   0   0   0   0   3
   0   0   0   0   0
   0   0   0   0   0

» A=diag(v,-2)
A =

   0   0   0   0   0
   0   0   0   0   0
   1   0   0   0   0
```

```
   0   2   0   0   0
   0   0   3   0   0
```

- creare matrici triangolari superiori o inferiori partendo da matrici assegnate, oppure prendere la parte superiore o inferiore a partire dalla k-sima sopra o sotto diagonale.

```
» A=[1 2 3; 4 5 6 ; 7 8 9]
A =

   1   2   3
   4   5   6
   7   8   9

» U=triu(A)
U =

   1   2   3
   0   5   6
   0   0   9

» L=tril(A)
L =

   1   0   0
   4   5   0
   7   8   9

» LL=tril(A,-1)
LL =

   0   0   0
   4   0   0
   7   8   0

» UU=triu(A,2)
UU =

   0   0   3
   0   0   0
   0   0   0
```

- L'operatore : che abbiamo già visto introducendo matrici e vettori serve anche per allineare tutti gli elementi di una matrice come vettore colonna. Vediamo con degli esempi

```
» A=[1 2 3; 4 5 6; 7 8 9]
A =

   1   2   3
```

```
    4    5    6
    7    8    9

» b=A(:)
b =

    1
    4
    7
    2
    5
    8
    3
    6
    9

» x=[1 2 3];
» x=x(:)
x =

    1
    2
    3

» x=[1; 2; 3]
x =

    1
    2
    3

» x=x(:)
x =

    1
    2
    3
```

L'incolonnamento è fatto secondo la memorizzazione della matrice che avviene colonna dopo colonna.

- Se si vogliono creare matrici vuote o eliminare elementi di una matrice, c'è l'operatore `[]` (le parentesi quadre chiuse).

```
» A=[]
A = [](0x0)
» A=[1 2 3; 4 5 6; 7 8 9]
A =

    1    2    3
```

```
    4    5    6
    7    8    9

» A(1,:)
ans =

    1    2    3

» A(1,:)=[]
A =

    4    5    6
    7    8    9
```

Nel primo caso siamo partiti da una matrice vuota. Nel secondo caso abbiamo una matrice con tre righe e tre colonne, piena, da cui eliminiamo la prima riga.

10.2 Operazioni tra matrici e vettori

Vediamo ora come gli operatori matematici che già conosciamo siano utilizzati per fare operazioni tra matrici e vettori

- L'operatore di moltiplicazione `*` serve per fare il prodotto tra matrici `A*B`, il prodotto matrice vettore (vettore colonna) `A*x`, il prodotto scalare tra due vettori (uno vettore riga e uno vettore colonna) `y*x`

```
» A=[1 2; 3 4]; B=[5 6; 7 8 ];
» C=A*B
C =

   19   22
   43   50

» x=[2;3];
» y=A*x
y =

    8
   18

» y=[1 2]; x=[3; 4];
» a=y*x
a =  11
```

- È possibile fare la trasposta di una matrice o di un vettore con l'operatore `'` (l'apostrofo)

```
» x
x =

   3
   4

» x'
ans =

   3   4

» A
A =

   1   2
   3   4

» A'
ans =

   1   3
   2   4
```

- L'operatore di divisione a sinistra \ permette di risolvere il sistema lineare $A\mathbf{x} = \mathbf{b}$

```
» A=[16 8 2; 8 29 1; 2 1 9.25]
A =

   16.0000    8.0000    2.0000
    8.0000   29.0000    1.0000
    2.0000    1.0000    9.2500

» b=[66 83  17.25]'
b =

   66.000
   83.000
   17.250

» x=A\b
x =

   3
   2
   1
```

Funzioni utili per risolvere problemi di algebra lineare sono

- **det** : calcola il determinante di una matrice
- **inv**: calcola la matrice inversa di quella data in input.
- **norm**: calcola la norma di un vettore o di una matrice. In particolare **norm**(A) o **norm**(A,2) danno la norma 2 della matrice `A`. Altre possibilità sono **norm**(A,1) (norma 1), **norm**(A, "inf") (norma infinito), **norm**(A, "fro") (norma di Frobenius).
- **lu**(A) produce la fattorizzazione LU della matrice `A`: [L,U]=**lu**(A) dà in output le matrici L e U tali che $LU = A$ con fattorizzazione di Doolittle, oppure se sono fatte permutazioni di righe, la matrice L è la matrice che comprende anche gli scambi di righe. [L,U,P]=**lu**(A) restituisce le matrici L e U triangolari inferiore e superiore, rispettivamente, insieme alla matrice di permutazione P tale che $PA = LU$.
- **chol** serve per la fattorizzazione di Cholesky: di default restituisce la matrice triangolare superiore U tale che $U^T U = A$. se si vuole la matrice triangolare inferiore L tale che $LL^T = A$, allora lo si deve specificare in input: U=**chol**(A) (primo caso) L=**chol**(A,'lower') (secondo caso)
- **eig** dà autovalori e autovettori della matrice: lambda=**eig**(A) dà solo gli autovalori; [v,lambda]=**eig**(A) dà autovettori e autovalori.

Una function importante per visualizzare come sono dislocati gli elementi di una matrice è data da **spy**. In genere viene applicata a matrici sparse (che hanno pochi elementi diversi da zero). E in genere, quando le matrici sono sparse, le si memorizza in un formato speciale in modo da non dover memorizzare tutti gli elementi uguali a zero della matrice stessa. Per trasformare una matrice dal formato pieno al formato sparso esiste la function **sparse**. Vediamo un esempio sull'uso di queste due funzioni.

Questo esempio ci mostra come usare le function che abbiamo appena visto per poter creare matrici tridiagonali.

```
» v=ones(10,1); A=diag(v)+2*diag(v(1:end-1),1)-4*diag(v(1:end-1),-1)
A =

   1   2   0   0   0   0   0   0   0   0
  -4   1   2   0   0   0   0   0   0   0
   0  -4   1   2   0   0   0   0   0   0
   0   0  -4   1   2   0   0   0   0   0
   0   0   0  -4   1   2   0   0   0   0
   0   0   0   0  -4   1   2   0   0   0
   0   0   0   0   0  -4   1   2   0   0
   0   0   0   0   0   0  -4   1   2   0
   0   0   0   0   0   0   0  -4   1   2
   0   0   0   0   0   0   0   0  -4   1
```

```
» A=sparse(A)
A =

Compressed Column Sparse (rows = 10, cols = 10, nnz = 28 [28%])

   (1, 1) ->  1
   (2, 1) -> -4
   (1, 2) ->  2
   (2, 2) ->  1
   (3, 2) -> -4
   (2, 3) ->  2
   (3, 3) ->  1
   (4, 3) -> -4
....
...(righe che non riportiamo)
   (10, 9) -> -4
   (9, 10) ->  2
   (10, 10) ->  1

» spy(A)
```

Il risultato della function **spy** è visibile in Figura 10.1.

Per passare da una matrice in formato sparso ad una piena, la function da applicare è **full**.

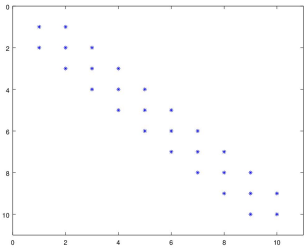


Figura 10.1: Funzione **spy**

10.4 APPLICAZIONI A VARI PROBLEMI

Vediamo come utilizzare al meglio le function predefinite di algebra lineare per risolvere semplici problemi di Calcolo Numerico. Lo facciamo risolvendo degli esercizi (e se dovessimo fare uso di function che non abbiamo ancora incontrato, le spiegheremo dove serve). Consideriamo la matrice

$$A = \begin{pmatrix} -2 & -5 & 6 \\ 0 & -1 & -2 \\ 1 & 3 & -1 \end{pmatrix}$$

Vogliamo calcolare la fattorizzazione LU tramite la function **lu**.

```
» A=[-2 -5 6; 0 -1 -2; 1 3 -1];
» [L,U]=lu(A)
L =

   1.00000   0.00000   0.00000
  -0.00000   1.00000   0.00000
  -0.50000  -0.50000   1.00000

U =

  -2  -5   6
   0  -1  -2
```

```
   0   0   1

» [L,U,P]=lu(A)
L =

   1.00000   0.00000   0.00000
  -0.00000   1.00000   0.00000
  -0.50000  -0.50000   1.00000

U =

  -2  -5   6
   0  -1  -2
   0   0   1

P =

Permutation Matrix

   1   0   0
   0   1   0
   0   0   1
```

In questo esempio, la matrice di permutazione è uguale all'identità, vale a dire che non sono stati fatti scambi di righe e colonne.

Vediamo un altro esempio:

```
» A=[1 3 -1; 0 -1 -2; -2 5 6];
» [L,U,P]=lu(A)
L =

   1.00000   0.00000   0.00000
  -0.50000   1.00000   0.00000
  -0.00000  -0.18182   1.00000

U =

  -2.00000   5.00000   6.00000
   0.00000   5.50000   2.00000
   0.00000   0.00000  -1.63636

P =

Permutation Matrix

   0   0   1
   1   0   0
   0   1   0
```

Nell'esempio appena fatto, invece, sono stati fatti scambi di righe.

In modo analogo si lavora se si vuole fare la fattorizzazione di Cholesky

```
» A=[16 -8 -6; -8 29 13; -6 13 42.25]
A =

   16.0000   -8.0000   -6.0000
   -8.0000   29.0000   13.0000
   -6.0000   13.0000   42.2500

» L=chol(A,'lower')
L =

    4.00000   0.00000   0.00000
   -2.00000   5.00000   0.00000
   -1.50000   2.00000   6.00000
```

Prendiamo ora spunto da esercizi proposti in temi d'esame.

Esercizio Si vuole approssimare il sistema lineare $A\mathbf{x} = \mathbf{b}$ con A matrice quadrata di dimensione $n = 30$ e $\mathbf{b}$ vettore assegnato, usando il metodo iterativo di Jacobi in modo da avere la soluzione approssimata con una tolleranza minore o uguale a 10^{-10}. Se la convergenza è lenta o non si arriva a convergenza, si ponga anche come numero massimo di iterazioni il valore 100. Il metodo di Jacobi va implementato in forma matriciale, partendo da un vettore iniziale $\mathbf{x}_0$ e applicando la formula $\mathbf{x}_{k+1} = E_J\mathbf{x}_k + \mathbf{q}$ fino a quando la norma euclidea dello scarto $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|$ non diventa minore della tolleranza prefissata. Si ricorda che $E_J = I - D^{-1}A$ e $\mathbf{q} = D^{-1}\mathbf{b}$ dove I è la matrice identità di dimensione n e D è la matrice diagonale con elementi diagonali uguali a quelli della diagonale principale di A.
La matrice A e il termine noto $\mathbf{b}$ vanno creati usando la function `Aebtema1.m` a disposizione, da usare all'interno dello script.
Si scriva, dunque, uno script che richiama le function `Aebtema1` e `jacobisimple`. Script e function siano strutturati nel modo seguente.
Per lo script:

1. definisca nella variabile `n` la dimensione richiesta nell'esercizio

2. generi la matrice A e il termine noto $\mathbf{b}$ richiamando la function a disposizione `Aebtema1` (si veda prima con l'help on line come utilizzarla)

3. definisca il vettore iniziale $\mathbf{x}_0$ come il vettore di elementi tutti nulli di lunghezza `n`

4. definisca una variabile per la tolleranza richiesta

5. definisca una variabile per il numero massimo di iterazioni consentite

6. chiami la function `jacobisimple` che in entrata deve avere come variabili `n`, `A`, `b`, e $\mathbf{x}_0$, la tolleranza e il numero massimo di iterazioni da poter effettuare, e in uscita deve avere il vettore approssimazione `x`, il numero di iterazioni effettuate `iter` e il vettore con la norma euclidea degli scarti ad ogni iterazione

7. faccia un grafico di convergenza semilogaritmico sull'asse delle ordinate che, in ascissa, riporti le iterazioni effettuate e in ordinata il valore degli scarti ad ogni iterazione

8. stampi il vettore che approssima la soluzione usando l'istruzione

```
fprintf(1,' vettore approssimazione ottenuto in %3i iterazioni \n', iter);
fprintf(1,'%14.10e \n ', x)
```

La function `jacobisimple` deve avere come variabili di ingresso la dimensione `n`, la matrice `A`, il vettore `b` e il vettore iniziale $\mathbf{x}_0$, la tolleranza e il numero massimo di iterazioni, e in uscita deve avere il vettore approssimazione `x`, il numero di iterazioni effettuate `iter` e il vettore con la norma euclidea degli scarti ad ogni iterazione. Quindi all'interno della function si deve costruire la matrice di iterazione $E_J = I - D^{-1}A$, il vettore $\mathbf{q} = D^{-1}\mathbf{b}$ e applicare il metodo iterativo di Jacobi fino a che la norma euclidea dello scarto tra due approssimazioni successive non diventa minore della tolleranza oppure si supera il numero massimo di iterazioni.
Per generare la matrice di Jacobi e calcolare la norma euclidea si usino function proprie di MATLAB®

La function che ci viene data per poter costruire la matrice e il vettore del sistema lineare da risolvere è fatta nel modo seguente

```
function [A,b]= Aebtema1(n)
% function per generare matrice e termine noto del sistema Ax=b
% function [A,b]= Aebtema1(n)
% in input il valore n del sistema da risolvere
% in output la matrice A e il vettore termine noto b
A=4*ones(n,1);
A=diag(A)- diag(ones(n-1,1),1) - diag(ones(n-1,1),-1);
b= sum(A,2);

end
```

La matrice A è una matrice tridiagonale, mentre il vettore $\mathbf{b}$ è dato dalla somma degli elementi di ciascuna riga di A, (il che significa che la soluzione del sistema sarà il vettore di tutti 1), dato che il prodotto della matrice A per un vettore di tutti 1 dà

come risultato un vettore che ha come componenti la somma degli elementi di ciascuna riga di A.

Scriviamo a questo punto lo script, passando in rassegna punto dopo punto ciò che ci viene chiesto nella traccia dell'esercizio. Dalla traccia, ciò che non conosciamo ancora sono le istruzioni relative a **fprintf** (da quello che leggiamo capiamo che serve per avere un certo formato nella visualizzazione dei risultati: vedremo dopo come usare al meglio questa function).

```
n=30;
[A,b]=Aebtema1(n);
x0=zeros(n,1);
tol=1.e-10; itmax=100;
[x,iter,vetscarti]=jacobisimple(n,A,b,x0,tol,itmax);
semilogy([1:iter],vetscarti)
fprintf(1,'␣vettore␣approssimazione␣ottenuto␣in␣%3i␣iterazioni␣\n', iter);
fprintf(1,'%14.10e␣\n␣', x)
```

Di per sè lo script è molto semplice. In effetti, il grosso del lavoro, spetta alla function che applica il metodo di Jacobi !!

```
function [xnew,iter,vettscarti]=jacobisimple(n,A,b,x0,tol,itmax)
% function per implementare lo schema di Jacobi
% in forma matriciale
% diag(A) da' il vettore degli elementi della diagonale principale di A
% diag(diag(A)) crea la matrice diagonale con gli elementi diagonali
% uguali a quelli di A
% creo la matrice inversa di questa
Dmenouno=inv(diag(diag(A)));
% creo la matrice di Jacobi applicando la formula che mi e' stata data
EJ=eye(n) -Dmenouno*A;
% creo il vettore q dello schema di Jacobi
q=Dmenouno*b;
% inizializzo a zero il vettore che conterra' la norma euclidea del vettore
% scarto ad ogni iterazione
vettscarti=zeros(itmax,1);
iter=0;
normascarto=2*tol;
xold=x0;
while normascarto >=tol && iter <itmax
    iter=iter+1;
    xnew= EJ*xold+ q;
    normascarto=norm(xnew-xold);
    vettscarti(iter)=normascarto;
    xold=xnew;
end
vettscarti=vettscarti(1:iter);
end
```

Come si vede, la struttura della function è molto simile a quella già vista nello schema di punto fisso o di Newton-Raphson. Ora però non stiamo lavorando con variabili scalari ma con vettori.

Quando eseguiamo lo script abbiamo

```
» esercjacobi
 vettore approssimazione ottenuto in  36 iterazioni
```

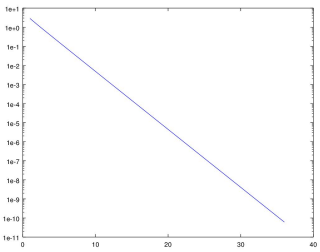


Figura 10.2: Grafico di convergenza semilogaritmico

```
1.0000000000e+00
 1.0000000000e+00
 9.9999999999e-01
 9.9999999999e-01
 9.9999999999e-01
 9.9999999999e-01
....
.... (righe che non riportiamo)
 9.9999999999e-01
 9.9999999999e-01
 1.0000000000e+00
 1.0000000000e+00
»
```

Il grafico di convergenza è riportato in Figura 10.2.

10.5 La function fprintf

La function che abbiamo appena usata nello script dell'esercizio precedente ci permette di scrivere con un certo formato le nostre variabili.

Questa operazione di scrittura può essere fatta sulla Command Window o su un file di scrittura dati.

Se rivediamo la prima istruzione abbiamo

```
fprintf(1,' vettore approssimazione ottenuto in %3i iterazioni \n', iter);
```

All'interno delle parentesi abbiamo un numero: il numero 1 è associato alla Command Window, quindi la scrittura dei dati viene fatta sulla Command Window (quello che abbiamo appena visto). Se mettiamo un altro numero, quel numero deve essere associato ad un file di scrittura (vedremo con un esempio come).

Poi abbiamo scritto tra apici una stringa di caratteri da visualizzare e poi il simbolo di percentuale % seguito da qualcosa %3i: quando leggiamo cosa viene stampato ci accorgiamo che quel %3i è associato alla variabile `iter` che è pure scritta all'interno di **fprintf**, una volta chiusi gli apici. Infine vediamo un \n (che non sappiamo cosa significhi).

Partiamo allora con ordine

- Per prima cosa decidiamo se dobbiamo stampare i risultati sulla Command Window o su file. Se dobbiamo stampare su Command Window ricordiamoci che dobbiamo scrivere 1 come prima cosa nella function **fprintf**. Se invece vogliamo stampare su file dobbiamo scrivere questo comando

  fid=**fopen**('nomefile', 'permesso');

- `fid` è la variabile che identifica il file e che sarà messo come prima cosa nella function **fprintf** (al posto dell'1 che identifica la Command Window)
- `fopen` è una function che serve per aprire il file con il nome che abbiamo scritto tra apici (per esempio `'dati.ris'`, `'risultati.txt'` ...)
- `'permesso'` è il permesso che diamo al file di scrittura/-lettura dati e può essere (vediamo le cose più importanti):
  * `'r'` diamo al file il permesso che possa essere letto
  * `'w'` diamo al file il permesso che possa essere scritto (se il file esiste già, sarà sovrascritto perdendo il contenuto che aveva)
  * `'a'` diamo al file il permesso che possa essere scritto appendendo ciò che si scriverà alla fine del file (se era stato già scritto): ‼c'è il rischio di creare file di lunghezza infinita se eseguiamo lo stesso script più e più volte.

- Il secondo passaggio è usare il comando **fprintf** per scrivere i dati

```
fprintf(fid, 'testo %formatovar1  altro testo \controllo', var1)
```

istruzioni per il formato		istruzioni di controllo	
istruzione	descrizione	istruzione	descrizione
%i oppure %d	formato intero	\n	nuova linea
%e oppure %E	formato scientifico	\r	andare a capo
%f	formato decimale	\b	cancellazione all'indietro
%g	formato più breve tra %e e %f	\t	tab
%s	formato di caratteri	''	apostrofo
		\\	\

Nel formato numerico possiamo mettere tra il simbolo di percentuale e il simbolo che indica il tipo di formato dei numeri che indicano le cifre da utilizzare: per esempio %3i vuol dire 3 cifre per numeri interi; %14.10e significa 14 cifre per il numero in formato scientifico, di cui 10 cifre riservate alla mantissa.

La variabile `var1` è una matrice (quindi una variabile scalare come caso particolare di matrice oppure un vettore o una matrice).

- Ultimo passaggio: una volta terminata l'operazione di scrittura, il file di scrittura va chiuso tramite l'istruzione

```
fclose(fid)
```

Vediamo degli esempi, ricordando (⚠cosa molto importante) che la funzione **fprintf** mostra la trasposta di una matrice.

Facciamo un esempio semplice, tenendo presente che se abbiamo variabili scalari da mostrare sulla stessa riga, allora le salveremo in un vettore all'interno della function **fprintf**.

```
fid=fopen('fileprova.txt','w');
a=1.2; b=2.5; intero=10;
stringa='prova';
fprintf(fid, 'questa e'' una %s \r', stringa);
fprintf(fid,'a e'' %g, b vale %5.2f  intero e'' %2i \n', [a b intero]);
fprintf(fid,'a e'' %5.2e, b vale %5.2f   \n', [a b]);
fprintf(fid, 'fine della %s \n', stringa);
fclose(fid);
```

Quando eseguiamo lo script viene generato il file `fileprova.txt` che vediamo in Figura reffileprova.

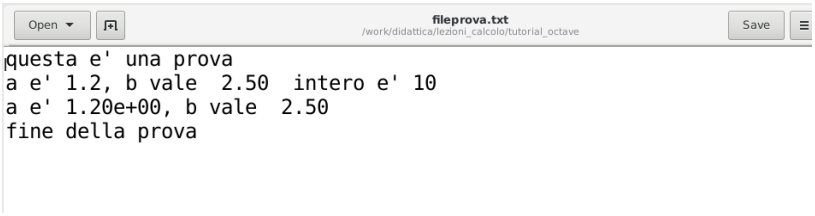


Figura 10.3: File `fileprova.txt`

Vediamo ora come visualizzare matrici, giocando sull'uso scorretto e corretto della loro rappresentazione (abbiamo detto che viene mostrata la trasposta della matrice con **fprintf**).

```
fid=fopen('altrofileprova.txt','w');
A=[1 2 3; 4 5 6];
B=[7 8 9; 10 11 12; 13 14 15];
fprintf(fid, 'uso errato per visualizzare matrici \n');
fprintf(fid,'matrice A %g %g %g \n', A);
fprintf(fid,'matrice B %g %g %g   \n', B);
fprintf(fid, 'uso corretto \n')
fprintf(fid, 'matrice A \n')
fprintf(fid, ' %g %g %g \n', A')
fprintf(fid, 'matrice B \n')
fprintf(fid, ' %2i %2i %2i \n', B')
fclose(fid);
```

Quando eseguiamo lo script, il file che viene generato ha questo contenuto:

```
uso errato per visualizzare matrici
matrice A 1 4 2
matrice A 5 3 6
matrice B 7 10 13
matrice B 8 11 14
matrice B 9 12 15
uso corretto
matrice A
 1 2 3
```

```
 4  5  6
matrice B
  7   8   9
 10  11  12
 13  14  15
```

10.6 UN ALTRA FUNCTION PER JACOBI

I metodi iterativi come quelli di Jacobi, Gauss-Seidel o del rilassamento sono molto lenti nelle applicazioni pratiche e superati da metodi più efficaci e robusti (come il metodo del gradiente coniugato e altri che vanno studiati in corsi più avanzatio di Calcolo Numerico). Un modo per implementare il metodo di Jacobi, tuttavia, può essere utile come strumento per aprire la strada a metodi più complicati. Prima abbiamo utilizzato la matrice del metodo facendo dei prodotti matrice-vettore utilizzando l'operatore di moltiplicazione $*$.

Ora vediamo un'altra modalità andando a tradurre in linguaggio di programmazione la formula che dà, componente per componente,

$$x_i^{(k+1)} = \overset{(D^{-1})_{ii}}{\Uparrow} \frac{1}{a_{ii}} \left[b_i - \overset{(L\mathbf{x}^{(k)})_i}{\Uparrow} \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \overset{(U\mathbf{x}^{(k)})_i}{\Uparrow} \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right] \qquad \text{per } i = 1, \ldots, n$$

L'approccio per aggiornare il vettore $\mathbf{x}^{k+1}$ (vale a dire `xold`) è basato proprio sulla formula scritta prima.

⚠Da notare che nella function che scriveremo ora, possiamo scegliere se scrivere o meno i risultati su un file di scrittura. Questa scelta fa sì che la function abbia un numero di variabili di input che può variare (a seconda che si dia o meno il nome del file di scrittura) Ciò comporta che, lavorando all'interno della function, si deve capire se è stato dato il nome del file su cui scrivere i risultati. Per capirlo si vanno a contare le variabili date in input e questo lavoro viene fatto dalla function predefinita **nargin**. Richiamata all'interno della function – dando il comando **nargin** – essa restituisce il numero di variabili date in input. Richiamata all'esterno di una function, essa restituisce il numero massimo di variabili che può avere la function (se scriviamo da Command Window **nargin**('jacobi') avremo come risultato 6.

Studiamo perciò con attenzione la function `jacobi.m` e tutti i commenti che vi abbiamo scritto.

```
function [xnew,iter,vettscarti]=jacobi(A,b,itmax,tol,x0,stringa)
% function [xnew,iter,vettscarti]=jacobi(A,b,itmax,tol,x0,stringa)
% implementazione del metodo di Jacobi per risolvere il sistema lineare
% Ax=b
% dati di input
```

```
% A : matrice quadrata del sistema lineare
% b : vettore termine noto
% itmax : numero massimo di iterazioni da effettuare
% tol : tolleranza sulla soluzione
% x0 : vettore di approssimazione iniziale
% stringa : nome del file di risultati, da scrivere come stringa di caratteri
%         esempio: stringa='risjacobi.ris'
%         la variabile stringa puo' anche essere omessa e, in tal caso,
%         non viene generato nessun file di risultati
% Quindi il numero di variabili date in input puo' essere o 6 (se diamo
% la stringa associata al file dei risultati, oppure 5
% scrivendo nargin all'interno della function il risultato sara' o 5 o 6
% Esempio
% A=[10 3 4; 2 8 5; 7 6 15];
% b=[17; 15; 28];
% itmax=200; tol=1.e-10;
% x0=zeros(3,1);
% [xnew,iter,vettscarti]=jacobi(A,b,itmax,tol,x0)
% si ricava
% xnew=
%    1.0000
%    1.0000
%    1.0000
% iter =  121
% e il vettore scarto di 121 componenti
%
% aggiungiamo la variabile
% stringa='risjacobi.ris';
% [xnew,iter,vettscarti]=jacobi(A,b,itmax,tol,x0,stringa)
% oltre ai dati di prima, viene generato il file risjacobi.ris
% da cui si puo' vedere come la stima del raggio spettrale
% sia 8.17131760e-01 e per la velocita' di convergenza si ha
% 8.77079092e-02
      m=size(A);
      if m(1)~=m(2)
              error('MATLAB:jacobi','matrice A rettangolare')
      end
      n=length(b);
      if m(1)~=n
          error('MATLAB:jacobi','matrice  e termine noto: diverse dimensioni')
      end
      vettscarti=zeros(itmax,1);
      scarto=2*tol;
      iter=0;
      xold=x0;
      xnew=zeros(n,1); % inizializzazione del vettore xnew per
                       % rendere piu' efficiente l'implementazione
      while iter<=itmax && scarto>=tol
         iter=iter+1;
         for i=1:n
             som1=A(i,1:i-1)*xold(1:i-1);
             som2=A(i,i+1:n)*xold(i+1:n);
             xnew(i) = (b(i)-som1-som2)/A(i,i);
         end
         sc=xnew - xold;
         scarto=norm(sc,2);
         vettscarti(iter)=scarto;
```

```
        xold=xnew;
    end
    vettscarti=vettscarti(1:iter);
% nargin e' una variabile "interna" che dice il numero delle variabili
% date in input alla function
    if nargin==6
        asint=scarto(2:iter)./scarto(1:iter-1);
        vel_conv=-log10(asint);
% vettscarti, asint e vel_conv sono vettori colonna, percio' li mettiamo
% come vettori riga nella matrice A che ci serve per la stampa
        A=[[1:iter]; vettscarti'; asint'; vel_conv'];
        fid=fopen(stringa,'w');
        fprintf(fid, '%5s %12s %12s %12s', 'iter', 'norma scarto', 'M', 'R');
        fprintf(fid, '\n%5d %12.8e %12.8e %12.8e', A);
        fprintf(fid, '\n \r');
        fclose(fid);
    end

    end
```

11

INTEGRAZIONE NUMERICA

10. Dà al mondo il meglio di te e sarai colpito fra i denti. DA' COMUNQUE IL MEGLIO DI TE.
Kent M. Keith (I dieci comandamenti paradossali)

UN ARGOMENTO MOLTO IMPORTANTE nel Calcolo Numerico è dato dall'integrazione numerica.

Vediamo subito come possiamo risolvere problemi di integrazione numerica, limitandoci al caso di funzioni scalari.

😱Il problema diventa più complicato quando si passa a funzioni che dipendono da due o tre variabili, come accade in molte applicazioni. Purtroppo non possiamo approfondire l'argomento ma è molto intrigante. 😀

11.1 FUNZIONI PREDEFINITE PER INTEGRARE

Partiamo dalle function predefinite che risolvono il problema dell'integrazione numerica.

Assegnato l'integrale $\int_a^b f(x)dx$, una function che possiamo utilizzare è la function **quad**

In MATLAB® questa function verrà sostituita in prossime versioni dalla function integral. Quindi se si lavora in MATLAB®, prestare attenzione ⚠.

Per lavorare con questa function, si deve indicare chi è la funzione f (tramite function handle) e chi sono gli estremi di integrazione. Eventualmente si può dare in input anche una tolleranza per la precisione del risultato.

Facciamo un esempio con $\int_2^5 \log(2+x)dx$. Di questo integrale sappiamo che una primitiva è $F(x) = (2+x)\log(2+x) - x$. Perciò il valore dell'integrale esatto è 5.0761935989 (dieci cifre decimali).

Per risolvere l'integrale dell'esempio, si fa integrazione per parti.

Applichiamo la function **quad**. E verifichiamo il risultato ottenuto.

```
f=@(x) log(2+x);
a=2; b=5;
F=@(x) (2+x).*log(2+x) -x;
Iex=F(b)-F(a);
I=quad(f,a,b);
err=abs(Iex-I);
disp([Iex I err])
```

Quando eseguiamo lo script (in format long) abbiamo come risultati:

```
» scriptquad
   5.07619359890763e+00   5.07619359890763e+00   1.77635683940025e-15
```

Per altre informazioni su questa function si rimanda all'help on line, da cui si traggono informazioni su altre function predefinite utili per integrare.

11.2 Formula dei trapezi e di Cavalieri-Simpson

Passiamo ora ad applicare le nostre conoscenze di base sulle formule di quadratura numerica per costruire delle function.

11.2.1 Trapezi

Per costruire una function che applichi la formula dei trapezi composta, eseguiremo due passaggi (data la funzione, gli estremi di integrazione e n il numero di suddivisioni):

1. costruiamo prima una function che applica la formula semplice
2. successivamente costruiamo una function che richiama la formula semplice su ogni suddivisione in modo da sommare i contributi su ogni suddivisione per ottenere l'integrale finale.

La formula semplice è molto *semplice* 😆😊

```
function  Itrap = trapsemplice(f,a,b)
%function  Itrap = trapsemplice(f,a,b)
% function per applicare la formula semplice dei trapezi
%  dati di input:  f funzione da integrare come function handle
%                  a  e b estremi dell'intervallo di integrazione
% dati di output: Itrap valore dell'integrale approssimato
Itrap= (b-a)*0.5*( f(a)+f(b) );
end
```

Quella composta richiede un po' di attenzione 🆘

```
function [ Itrap,h ] = trapcomposta(f,a,b,n)
% function [ Itrap,h ] = trapcomposta(f,a,b,n)
% function per applicare la formula composta dei trapezi
%  Itrap= somma della formula dei trapezi semplice su ciascun
%         sottointervallo
%  dati di input:  f funzione da integrare
%                  a  e b estremi dell'intervallo di integrazione
%                  n  numero di suddivisioni
% dati di output: Itrap valore dell'integrale con la formula dei trapezi
%                 h ampiezza delle suddivisioni
h=(b-a)/n;
x0=a; % primo estremo di integrazione
Itrap=0;
for i=1:n
x1=x0+h; % secondo estremo di integrazione sull'i-sima suddivisione
Itrap= Itrap+ trapsemplice(f,x0,x1);
x0=x1; %aggiornamento degli estremi di integrazione
end
end
```

La function `trapcomposta` dà in input sia il valore dell'integrale approssimato, sia l'ampiezza delle suddivisioni.

Risolviamo il problema precedente con la formula dei trapezi e considerando diverse suddivisioni.

⚠La function `trapcomposta` dà due variabili in uscita, per come l'abbiamo scritta. Tuttavia, se si scrive `I=trapcomposta(f,a,b,n)` in uscita abbiamo solo il valore dell'integrale e si perde il valore di `h`, che abbiamo invece scrivendo `[I,h]=trapcomposta(f,a,b,n)`

```
f=@(x) log(2+x);
a=2; b=5;
F=@(x) (2+x).*log(2+x) -x;
Iex=F(b)-F(a);
nvett=[1 2 4 8];
indice=0;
% costruiamo due vettori, Itrap e h con i valori dell'integrale
% approsimato e del corrispondente valore di h per i valori di
% n = 1 2 4 e 8 (cosi' come inseriti nel vettore nvett)
% dal momento che sono solo 4 componenti non e' necessario inizializzare i
% vettori Itrap e h.
% Importante e' creare vettori le cui componenti siano indicizzate
% con valori che partono da 1 e poi incrementano di 1 il valore
% delle componenti (come fa la variabile indice)
for n=nvett
  indice=indice+1;
  [Itrap(indice), h(indice)]=trapcomposta(f,a,b,n);
end
err=abs(Iex-Itrap); %err e' un vettore
disp(Itrap)
disp(err)
subplot(2,1,1)
semilogy(h,err)
xlabel('ampiezza h')
ylabel('errore assoluto')
subplot(2,1,2)
plot(nvett(2:end), err(1:end-1)./err(2:end))
xlabel('livello suddivisioni')
ylabel('rapporto errori')
```

Se eseguiamo lo script i risultati danno errori molto grandi rispetto alla function predefinita, (d'altra parte dalla teoria sappiamo che la formula dei trapezi è molto grossolana).

```
» scripttrap
   4.99830676526281   5.05627552098904   5.07118232184066   5.07493871469654
 Columns 1 through 3:

   0.07788683364482463   0.01991807791858857   0.00501127706696547

 Column 4:

   0.00125488421108777
```

Ma la cosa bella è che tutto quello che abbiamo studiato nella teoria su questo metodo, lo ritroviamo. Basta vedere in Figura 11.1 come l'errore decresce al diminuire dell'ampiezza h (il grafico di convergenza va visto da destra verso sinistra) e come il rapporto degli errori tenda ad un valore ben preciso (4).

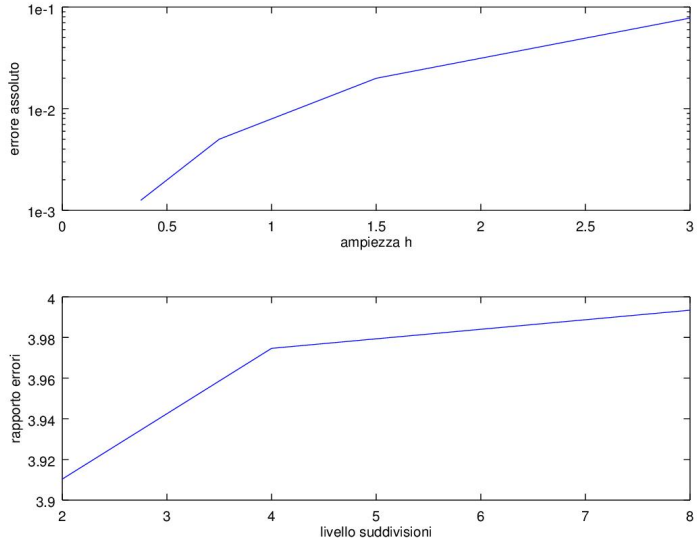


Figura 11.1: Applicazione del metodo dei trapezi

11.2.2 CAVALIERI-SIMPSON

Stesso discorso si può ripetere per la formula di Cavalieri-Simpson.

Scriveremo due function: una che applica la formula semplice e l'altra che richiama la formula semplice su ogni suddivisione. Per analogia chiamiamo le due function `cavsemplice.m` e `cavcomposta.m`. In realtà è la function semplice che richiede qualche modifica, perchè la function per applicare la formula composta sarà praticamente la stessa a quella che abbiamo già scritto ma con il nome diverso della function che viene richiamata all'interno. Difatti, se stiamo applicando la formula semplice, possiamo dare in input sempre e solo gli estremi di integrazione e il punto medio dell'intervallo lo costruiamo all'interno della function.

Vediamo bene

```
        function  Icav = cavsemplice(f,a,b)
%function  Icav = cavsemplice(f,a,b)
% function per applicare la formula semplice di Cavalieri-Simpson
%  dati di input:  f funzione da integrare come function handle
%                  a  e b estremi dell'intervallo di integrazione
%  dati di output: Itrap valore dell'integrale approssimato
c=(a+b)/2;
Icav= (b-a)*(f(a)+4*f(c)+f(b) )/6;
end

function  [Icav,h] = cavcomposta(f,a,b,n)
% function  [Icav,h] = cavcomposta(f,a,b,n)
% function per applicare la formula composta di Cavalieri-Simpson
%  dati di input:  f funzione da integrare
%                  a  e b estremi dell'intervallo di integrazione
```

```
%                    n  numero di suddivisioni
%  dati di output: Icav valore dell'integrale
%                   h ampiezza delle suddivisioni
h=(b-a)/n;
x0=a;
Icav=0;
for i=1:n
x1=x0+h;
Icav= Icav+cavsemplice(f,x0,x1);
x0=x1;
end
end
```

Osserviamo come la function `cavcomposta` cambia rispetto alla `trapcomposta` nel nome delle variabili: dove prima avevamo scritto `trap` ora abbiamo scritto `cav`. Possiamo quindi

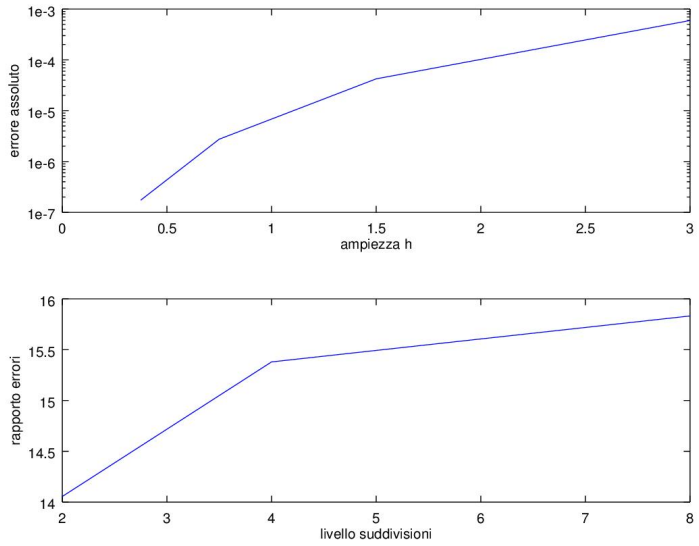


Figura 11.2: Applicazione del metodo di Cavalieri-Simpson

scrivere uno script del tutto analogo a quanto abbiamo fatto per i trapezi, per risolvere lo stesso problema. Riportiamo i risultati:

```
» scriptcav
   5.07559843956445   5.07615125545787   5.07619084564850   5.07619342499655
 Columns 1 through 3:

   5.95159343177443e-04   4.23434497562880e-05   2.75325912912905e-06

 Column 4:

   1.73911084644374e-07
```

I risultati sono in linea con la teoria studiata (vedasi Figura 11.2).

12

Informazioni utili

Che cos'è la vita? È il lampo di una lucciola nella notte. È ' il respiro di un bufalo nel periodo invernale. È la piccola ombra che corre attraverso l'erba e si perde nel tramonto.
Crow Foot

Concludiamo queste note con alcuni dettagli che non hanno trovato spazio nelle pagine precedenti.

12.1 Sulla function input

Quando usiamo la function **input** in genere facciamo qualcosa del genere
variabile=**input**('scrivi␣il␣valore␣numerico␣della␣variabile:␣'); Perciò dalla Command Window, scriviamo un numero.

A volte però si vuole dare in input una stringa di caratteri. Per fare ciò dobbiamo aggiungere un'altro dato di input alla function **input**!. Vediamo con un esempio.

Abbiamo un file di dati dal nome `dati.dat` e vogliamo assegnare il nome ad una stringa di caratteri. Faremo qualcosa del genere:

```
filename=input('scrivi il nome del file: ', 's');
A=load(filename)
```

In questo modo noi dobbiamo scrivere il nome del file senza apici!

```
» scriptinputstringa
scrivi il nome del file: dati.dat
A =

    1.000000000000000    0.500000000000000
    2.000000000000000    1.400000000000000
    3.000000000000000    3.700000000000000
    4.500000000000000    8.100000000000000
    5.000000000000000   10.400000000000000
    6.000000000000000   13.699999999999999
    7.500000000000000   21.899999999999999
    8.000000000000000   24.600000000000001
    9.100000000000000   32.399999999999999
```

Avete notato cosa succede se non lasciamo un po' di spazio prima di chiudere la stringa di caratteri della frase che scriviamo sulla Command Window? Provate a scrivere
var=**input**('scrivi␣var:');
e poi var=**input**('scrivi␣var:␣');
Riprendiamo l'esempio sulla conversione di gradi Celsius, Fahrenheit e Kelvin, visto nel Capitolo 5 dove avevamo usato stringhe di caratteri da dare in input.

```
   10.000000000000000   40.299999999999997

»
```

La stringa `'s'` inserita come seconda variabile alla function **input** ci permette di fare questo. Se non mettiamo `'s'`, per poter assegnare la variabile stringa, dobbiamo scrivere il nome del file tra apici. In questo caso, però, sarebbe meglio ricordare che bisogna usare gli apici, in modo da evitare errori!

```
filename=input('scrivi il nome del file: ');
A=load(filename)
```

In tal caso scriveremo

```
scrivi il nome del file: 'dati.dat'
A =

    1.000000000000000    0.500000000000000
    2.000000000000000    1.400000000000000
         etcetera etcetera
```

12.2 DATA E ORA

Se nell'esecuzione dei nostri script vogliamo inserire la data e l'ora dell'esecuzione ci sono diverse possibilità. Ne indichiamo due

```
» date
ans = 16-Feb-2017
» disp(date)
16-Feb-2017
» datestr(now)
ans = 16-Feb-2017 11:11:28
» disp(datestr(now))
16-Feb-2017 11:11:31
» now
ans =    7.3674e+05
```

La function **date** mostra la data corrente (giorno, mese, anno). La function now oltre alla data ci dà informazioni anche sull'ora: scrivere però now non ci dice nulla perchè dà il *serial day number*. Con datestr convertiamo il *serial day number* in un formato a noi comprensibile.

12.3 TEMPI DI ESECUZIONE

Due importante function sono **tic** e **toc** che danno i tempi di esecuzione delle righe di programma scritte tra **tic** e **toc**.

In questo modo si possono confrontare tempi di esecuzione.

Vediamo un esempio andando a creare un vettore con molte componenti, usando diversi approcci.

```
clear
n=10^6;
a=1;b=2;
% creiamo il vettore x di n componenti equidistanti tra a e b
% con la function linspace
tic
x=linspace(a,b,n);
toc
% creiamo l'analogo vettore che chiamiamo y usando un ciclo for
% e preallocando le componenti del vettore
tic
h1=(b-a)/(n-1);
y=zeros(n,1);
for i=1:n
    y(i)=a+h1*(i-1);
end
toc
% creiamo il vettore w (con le stesse componenti degli altri due vettori)
% usando l'operatore :
tic
h2=(b-a)/(n-1);
w=a:h2:b;
toc
```

```
» scripttictoc
Elapsed time is 0.000167131 seconds.
Elapsed time is 0.579937 seconds.
Elapsed time is 1.69277e-05 seconds.
```

Dai risultati dell'esecuzione, vediamo come il ciclo `for` sia da evitare se ci sono altre strade perchè è quello che impiega più tempo. Se modifichiamo il codice e mettiamo $n = 10^6$ al posto di $n = 10^5$ abbiamo

```
» scripttictoc
Elapsed time is 0.00349903 seconds.
Elapsed time is 5.73008 seconds.
Elapsed time is 1.69277e-05 seconds.
```

Le differenze dei tempi di esecuzione si notano maggiormente! 😱

⚠In MATLAB®lo stesso script produce differenze meno esagerate ma il ciclo `for` ne esce sempre sconfitto (anche se di poco). Le differenze tra un software e l'altro sono dovute ai diversi codici che gestiscono le stesse strutture. Per $n = 10^6$, MATLAB®ci dà

```
» scripttictoc
Elapsed time is 0.007866
seconds.
Elapsed time is 0.009908
seconds.
Elapsed time is 0.004055
seconds.
```

Bibliografia

AA.VV. *Gnu Octave*. Documentazione on line, https://www.gnu.org/software/octave/doc/interpreter/.

Tobin A. Driscoll. *Learning MATLAB®*. SIAM, 2009.

Amos Gilat. *MATLAB® An Introduction with Applications*. John Wiley & Sons, Inc., 2011.

William J. Palm III. *Introduction to MATLAB® for Engineers*. McGraw-Hill, 2011.

Jerome Lecoq. *Learning MATLAB® for new and advanced users*. Blog in internet, http://www.matlabtips.com/.

Annamaria Mazzia. *Laboratorio di Calcolo Numerico, Applicazioni con MATLAB® e Octave*. Pearson, 2014.

David McMahon. *MATLAB® Demystified, A self-teaching guide*. McGraw-Hill, 2007.